Radboud University Nijmegen

Proving the power of Synthesis Based Engineering on the TNO-ESI Demo system

Author:
Cosmin Bacau
s1070065
cosmin.bacau@ru.nl

First supervisor/assessor:
dr. ir. Dennis Hendriks
dennis.hendriks@ru.nl

TNO supervisor:
dr. Andrea Peruffo
andrea.peruffo@tno.nl

Second assessor:
dr. Frits Vaandrager
f.vaandrager@cs.ru.nl

Contents

Abstract

This project investigates the application of Synthesis-Based Engineering (SBE) to the TNO Embedded Systems Innovation (TNO-ESI) demo system, a modular cyber-physical training factory. The objective is to demonstrate the power of SBE in the development of industrial control software by automatically synthesizing correct-by-construction supervisory controllers from formal specifications. To evaluate the applicability and expressive power of SBE, four demonstration scenarios are developed. These demonstrations focus on system functionality: a single-product workflow scenario, coordinated two-product handling, a step-by-step scenario that swaps two products, and a demonstration that again swaps two products, but only the desired outcome is specified, while the synthesis algorithm autonomously determines how this action happens. These demonstrations validate the correctness of the synthesized controller under increasing system complexity and illustrate how SBE supports systematic extension of system behavior by modifying specifications rather than control code. They also demonstrate how SBE enables the realization of complex behavior with minimal engineering effort while preserving correctness guarantees by construction. The project consists of work across the SBE workflow, including system and requirements modeling using formal specification languages, synthesis of supervisory controllers, visualization of system behavior and synthesized controllers and code generation for deployment on the physical TNO-ESI demo system. This application demonstrates SBE as a viable approach for developing reliable industrial control systems with reduced manual programming effort and built-in correctness guarantees.

1 Introduction

In contemporary engineering, automation has become a fundamental aspect of cyber-physical systems. As the complexity of these systems increases, it is important to address this complexity by using methods that help engineers design control behavior in a structured way. One key approach to achieve such automation is through Synthesis-Based Engineering (SBE). This engineering methodology relies on creating clear, structured models of a system so that its behavior is correct from the start [7]. By specifying what the system should do, rather than how to implement it, engineers can verify correctness at the model level before any code is written. This provides mathematical guarantees about system behavior. The result is systems that are correct-by-construction relative to their specifications. In this approach, the goal is to prevent unwanted and unsafe behavior, by guiding the system through well defined rules. The system is designed in such a way that only the intended and safe actions are possible in the first place, ensuring that it operates according to the purpose and needs for which it is designed. In SBE, the first step is to describe the physical behavior of a system. After that, the next step is to define a set of rules such that the system can operate in a safe and efficient manner. At the core of SBE lies the construction of formal system models, which form the foundation of the entire development process. In traditional engineering, the requirements are collected separately in a document, then translated into code and later the implementation is tested. On the other hand, SBE uses these formal models in order to directly generate the supervisory logic.

This thesis explores how Synthesis-Based Engineering (SBE) can be applied to the TNO-ESI demo system, a small industrial setup designed for research and demonstration purposes. The demo system provides a practical environment where the principles of SBE, such as modeling system behavior, defining formal requirements, and synthesizing supervisory controllers, can be tested and demonstrated in a clear and visual way. By using this system, we ain a clearer understanding of how SBE works in practice and what unique advantages it offers.

The primary question guiding this work is:

How can we demonstrate the unique values of SBE using the TNO-ESI demo system, making them tangible and visible?

To systematically address this question, five sub-research questions are formulated, each aligned with key steps in the SBE process:

SRQ1: How can we best model the plants and requirements of the TNO-ESI demo system?

SRQ2: How can we best develop a simulation model for the TNO-ESI demo system to validate the plants and requirements?

SRQ3: How can we efficiently synthesize a supervisor for the TNO-ESI demo system?

SRQ4: How can we best connect the synthesis result for the TNO-ESI demo system to the digital twin and the physical system?

SRQ5: What demonstrations can we create to best showcase the value of the SBE methodology using the TNO-ESI demo system?

To answer these questions, we follow a structured process consisting of six main steps:

1. define and model the system,
2. model the requirements,

3. synthesize the logic that controls the system,

4. use simulation to test our implementation,

5. generate the code in Java,

6. create demos that prove the power of SBE.

The first step (part of SRQ1) involves defining and modeling the TNO-ESI demo system's behavior without any control mechanism applied, capturing it in a clear and understandable formal model. In this work we use the CIF modeling language, provided by the Eclipse ESCET open source project, as described in Section 2.2. The second step (part of SRQ1) focuses on formulating a specific set of requirements that ensures proper system functioning. These requirements establish constraints on system behavior, for example preventing physical collisions or undesired actions. The third step (SRQ2) consists in developing a simulation model with visualization to validate the designed plants and requirements. A 2D representation resembling the actual 3D demo system allows us to observe system behavior and verify that the modeled behavior is correct. Furthermore we can also validate that we have the desired requirements. This validation step helps to identify whether adjustments are needed. If unexpected behavior occurs during simulation, the plant and requirements can be immediately modified to obtain the correct system behavior. Based on the validated model and requirements, the fourth step (SRQ3) is to synthesize the supervisory logic that guides the system to operate accordingly to our needs. This synthesis process generates a controller that is guaranteed to be correct by construction with respect to the specified requirements. The fifth step (SRQ4) involves generating executable Java code from the formal specification and integrating it with the demo system's software configuration. The generated code is adapted to interface with both the physical system and its digital twin, ensuring seamless deployment. Finally, the sixth step (SRQ5) is to design and implement demonstrations that showcase the tangible benefits of SBE, illustrating the unique values of the approach.

The remainder of this thesis is structured as follows:

Chapter 2 provides preliminaries on the SBE methodology, introduces the CIF language and toolset, and discusses the advantages of SBE in industrial automation, establishing the theoretical background for this work.

Chapter 3 reviews related work in the field of supervisory control and formal methods applied to industrial systems.

Chapter 4 presents a comprehensive overview of the TNO-ESI demo system, including its physical components, the digital twin, and the communication architecture including the interface between the controller and the system.

Chapter 5 (SRQ1, Step 1) details the development of the formal CIF model of the system, describing how each component is modeled as automata.

Chapter 6 (SRQ2, Step 3) describes the simulation model development, focusing on a visualization-based means of validation.

Chapter 7 (SRQ1, Step 2) presents the formal requirements specification, including both safety requirements that prevent hazardous behavior and scenario requirements that define desired operational sequences.

Chapter 8 (SRQ3, Step 4) discusses the supervisor synthesis process, explaining the use of Binary Decision Diagrams (BDDs) and configuration settings used to tune the use of BDDs for better synthesis performance.

Chapter 9 (SRQ4, Step 5) details the code generation process, describing how the synthesized controller is translated into executable Java code and integrated with the demo system's software infrastructure.

Chapter 10 (SRQ5, Step 6) presents the demonstrations and results, showcasing functional demos that illustrate the practical value and unique benefits of the SBE approach on the TNO-ESI demo system.

Chapter 11 concludes the thesis by reflecting on how the research questions have been answered, summarizing key findings, and discussing directions for future work.

2 Preliminaries

2.1 Synthesis Based Engineering as methodology

Synthesis-Based Engineering is a model-driven engineering methodology that bridges the gap between high-level system specifications and the automatic construction of correct control logic for cyber-physical systems. Rooted in the theory of supervisory control of discrete-event systems, SBE provides a mathematically rigorous framework in which system behavior and control objectives are expressed formally and used as the basis for controller synthesis. SBE integrates formal modeling, validation, synthesis, and code generation into a single workflow that yields correct-by-construction controllers [7].

The central idea behind SBE is that system behavior should not be programmed directly, but instead derived automatically from a precise specification of what the system must do. This synthesis process guarantees several fundamental correctness properties:

- **safety** ensures that forbidden or hazardous system states are never reached
- **non-blockingness** guarantees that the system can always reach a marked state
- **controllability** ensures that the controller refrains from disabling events beyond its control
- **maximal permissiveness** ensures that the controller restricts behavior only when strictly necessary, allowing all behavior that does not violate the requirements

Together, these properties ensure that the synthesized controller is correct for the given plant and requirements.

2.2 The CIF Language

This section introduces the fundamental CIF concepts required to understand the modeling, simulation, synthesis, and deployment steps presented in the remainder of this thesis. All subsequent CIF models build upon these notions of automata, invariants, events, synchronization, and the separation between plant and requirement models.

CIF [2] is a formal modeling language that is used to model the system under consideration by explicitly separating plant behavior from control requirements, to represent synthesized supervisors, and to define simulation and visualization models. In addition to being a modeling language, CIF is also a comprehensive toolset that supports the entire SBE workflow, including modeling, validation through simulation, supervisory controller synthesis, and code generation.

CIF is developed as part of the Eclipse ESCET[1] open-source project, which provides tool support for model-based engineering of supervisory control systems [5]. The ESCET toolset offers facilities for editing CIF models, simulating system behavior, synthesizing supervisors, and deploying synthesis results to software implementations. More information about ESCET can be found on the website: `https://eclipse.dev/escet/`.

The core modeling concept in CIF is represented by automata. "Automata are abstract models of machines that perform computations on an input by moving through a series of states or configurations" [1]. In this thesis, automata are used to model the behavior of the TNO-ESI demo system.

[1] 'Eclipse', 'Eclipse ESCET' and 'ESCET' are trademarks of Eclipse Foundation, Inc.

An automaton is composed of a finite set of locations and transitions between those locations. A location represents a particular state or configuration of the system. At any point in time, an automaton is in exactly one location, representing its current state. Each automaton has one or more *initial* locations, which define where the automaton can start when the system is initialized. CIF also uses *marked* locations, which represent goal states or desired completion points that the system should be able to reach. The transition from one state to the other is done by using edges, where each edge points from a source location to a target location. Moreover, these edges are labeled with events, which represent something that happens in the system, such as an action being performed or a change occurring. Events are the triggers that cause the automaton to transition from one location to another. For instance, a "*button_pressed*" event might cause a door automaton to transition from a "closed" location to an "open" location. When an event occurs, the automaton follows the corresponding edge to move to a new location. The events are split into two categories: *controllable* events that are an action the system can perform that the controller can also disable, such as turning an actuator on or off, and *uncontrollable* events that cannot be disabled by the controller and act independently. For example, an uncontrollable event could be the reading of a sensor value, which reflects the physical state of the system regardless of the controller's actions [3]. Edges may additionally be equipped with guards, which are boolean expressions that must evaluate to true for the edge to be enabled, and updates, which modify variable values when the transition is taken (for example, $x := 5$ or $x := x + 1$). Guards and updates allow automata to capture data-dependent behavior.

CIF supports synchronization between automata through shared events. When multiple automata share an event, they must all participate in the corresponding transition simultaneously. For such a synchronized transition to occur, every participating automata must have an enabled edge labeled with that event from its current location. When the event occurs, all automata move to their respective target locations, and all updates are applied atomically. CIF follows the global read, local write principle: all automata may read any variable, but only the automaton that declares a variable is allowed to update it.

In addition to transitions, CIF provides *invariants*, which are conditions that must hold continuously while the system is in a particular location. Invariants are commonly used to model safety constraints or physical limitations, such as preventing a resource from exceeding its capacity or ensuring that a mechanism does not remain in an unsafe state. Furthermore, they play an important role in constraining system behavior. There are two types: *state requirement invariants* that specify predicates which must hold in all reachable states, ensuring that any violating states are classified as undesirable and prevented by synthesis. The other type of invariants are *state/event exclusion requirement invariants* which specify conditions that determine when events are enabled or disabled in certain states. Invariants are commonly used to model safety constraints or physical limitations, such as preventing incompatible states (e.g., a bridge deck being open while a traffic light is green) or restricting transitions that lead to unsafe conditions.

Within CIF, every automaton and invariant is explicitly classified as either a *plant* or a *requirement*. Plant automata and invariants describe the uncontrolled behavior of the system and therefore define everything that is physically possible, including unsafe behavior. Requirement automata and invariants impose constraints on this behavior, excluding undesired or unsafe executions and ensuring correct system operation. From these plant and requirement models, CIF's synthesis algorithm automatically generates

a *supervisor*, a controller that restricts the plant's behavior by selectively disabling controllable events to ensure that all requirements are satisfied. The supervisor operates by monitoring the current state and preventing transitions that violate any requirement, thereby guaranteeing safe and correct system operation. CIF also supports *input variables*, which represent values originating from the environment, such as sensor readings. Input variables can be read anywhere in the model but cannot be assigned to, as their values are controlled externally and may change independently of the controller.

To support modular modeling, CIF distinguishes between automaton definitions and automaton instantiations. An automaton definition specifies the structure and behavior of an automaton in an abstract, reusable form. It may include parameters that allow the same definition to be instantiated multiple times with different configurations. An automaton instantiation creates a concrete instance of such a definition within the model, providing arguments for each of the parameters of the definition This mechanism enables compact modeling of systems that contain multiple similar components, while maintaining consistency and reducing duplication.

CIF also provides groups, which allow automata and invariants to be organized into logical units. Groups improve the readability and maintainability of complex models by bundling related components together.

Finally, CIF allows simulation by using SVG images. In this case, users have the option of urgent and non-urgent events, which control the timing behavior of transitions in the automaton. Non-urgent events means the system is not forced to execute those events immediately when they become enabled. Non-urgent events allow the automaton to remain in its current location even when a transition is available, providing flexibility in the timing of state changes. This is particularly useful for modeling situations where an action may occur at any time within a certain period, rather than being forced to happen instantaneously. Urgent events, in contrast, must be executed as soon as they become enabled. When a transition labeled with an urgent event becomes available, the system cannot let time pass in the current state, as it must take the transition immediately. This enforces instantaneous behavior and is useful for modeling immediate reactions. The distinction between urgent and non-urgent events is essential for understanding where some actions have strict timing requirements while others can occur with more flexibility.

2.3 Advantages of SBE in Industrial Automation

In modern automation, SBE changes the traditional approach of designing first and checking later. Instead, engineers specify what they *want* the system to do (the desired behavior and properties), rather than *how* to implement it (the detailed control logic), and the correct controller is automatically created. Engineers model system behavior using automata that show different states and actions, rather than writing specifications, coding the implementation manually, and then testing everything afterward. Applying Synthesis-Based Engineering within an industrial setting yields several advantages, most notably guaranteed correctness. Supervisory synthesis produces a controller that is correct-by-construction, meaning that it enforces both safety and liveness properties. Safety properties specify that the system must never reach forbidden or unsafe states, whereas liveness properties guarantee that the system can always make progress toward desired states and does not become permanently blocked [5], [6], [7]. Furthermore, the development time is reduced because the automation of controller synthesis frees engineers from manually implementing the logic of the system. SBE naturally supports

subsystem-level modeling, enabling plant automata to be developed independently and integrated via synchronization providing a modular and scalable modeling. Moreover, because the controller is derived directly from formal specifications, every behavior is traceable to an explicit requirement improving long-term maintainability [5], [6]. Unlike traditional engineering, where requirements are written in natural language, collected in documents that are often ambiguous and open to multiple interpretations, and then manually translated into code, Synthesis-Based Engineering (SBE) enforces formality, automation, traceability, and correctness-by-construction. In SBE, engineers expresses all specifications mathematically, computes supervisors algorithmically, guarantees that every system behavior corresponds to a modeled requirement, and enables the automatic generation of controller code in multiple industrial programming languages, ensuring consistency between simulation and real-world deployment. Lastly, SBE enables automatic code generation for deployment on industrial systems, ensuring that simulated behavior translates directly to real-world operation.

3 Related Work

Several studies have already explored different applications of Synthesis-Based Engineering across multiple domains, as systematically reviewed [4]. That work presents, in Chapter 8 of the literature review, the growth of applications in areas such as robotics, manufacturing systems, autonomous driving systems, and infrastructural systems.

Notable contributions in the domain of manufacturing systems include structured engineering approaches for model-based supervisor design, demonstrated through a container-handling system incorporating load platforms, truck cranes, and storage facilities [16]. Furthermore, applications in semiconductor manufacturing have addressed deadlock elimination and behavioral optimization through state trimming techniques [10].

The robotics domain has experienced considerable advancement in supervisory control applications, particularly in autonomous navigation systems. Navigation scenarios involving unpredictable obstacles have motivated the use of multi-supervisor architectures, in which control responsibilities are distributed between path execution and secondary requirements such as obstacle avoidance [8]. Swarm robotics applications have employed supervisory control theory to formalize inter-robot coordination and communication protocols [14], with extensions addressing human-operator interaction within swarm control hierarchies.

In the domain of autonomous driving systems, supervisory synthesis has proven instrumental in ensuring correct and safe execution of autonomous driving functions. In particular, lane-changing modules have been studied, where errors in manually designed supervisors are identified and resolved through synthesis-based approaches [13]. Moreover, the coordination of advanced driver assistance systems (ADAS), including lane-changing modules, cruise controllers, and collision-avoidance systems, has been achieved using supervisors synthesized from multiple plant models and requirement specifications within sub-second computation times [12].

Lastly, Synthesis-Based Engineering has also been demonstrated as an effective methodology in infrastructural systems. For example, lock-bridge systems of substantial complexity controlling dozens of actuators and sensors with extremely large state spaces have been successfully supervised using symbolic synthesis techniques and extensive requirement modeling [15].

TNO-ESI is a supporter of SBE, but the researchers do not always have direct access to partner systems for hands-on experimentation and educational purposes. Having a dedicated demo system at TNO-ESI addresses several critical needs: it provides students and researchers with a tangible platform to learn and experiment with SBE methodologies. Most importantly, it serves as a practical demonstrator to showcase the value of methods and tools developed at TNO-ESI such as SBE in a concrete and accessible way. This makes the TNO-ESI demo system not merely another application of SBE, but a strategic asset for education, research, and industrial engagement.

4 TNO-ESI Demo System

The TNO-ESI demo system is a commercially available Fischertechnik Training Factory Industry 4.0 platform [18], and constitutes a cyber-physical platform designed to emulate the structure of a generic automated factory. In addition to the physical hardware, the TNO-ESI demo system includes a digital twin, a lower-level PLC-based control implementation, and an Industrial Hardware Abstraction Layer (IHAL). These software components are developed by ICT Group, another research unit at TNO. Together, the physical system and its software counterpart represent a good medium to study supervisory control strategies in a realistic industrial context.

The TNO-ESI demonstration platform (Figure 1) and the digital twin (Figure 2) consist of four main subsystems: *warehouse*, *gripper robot*, *multi-processing station* and *sorting line* that together represent the stages of a simplified production chain: storage, transport, processing, and final sorting.

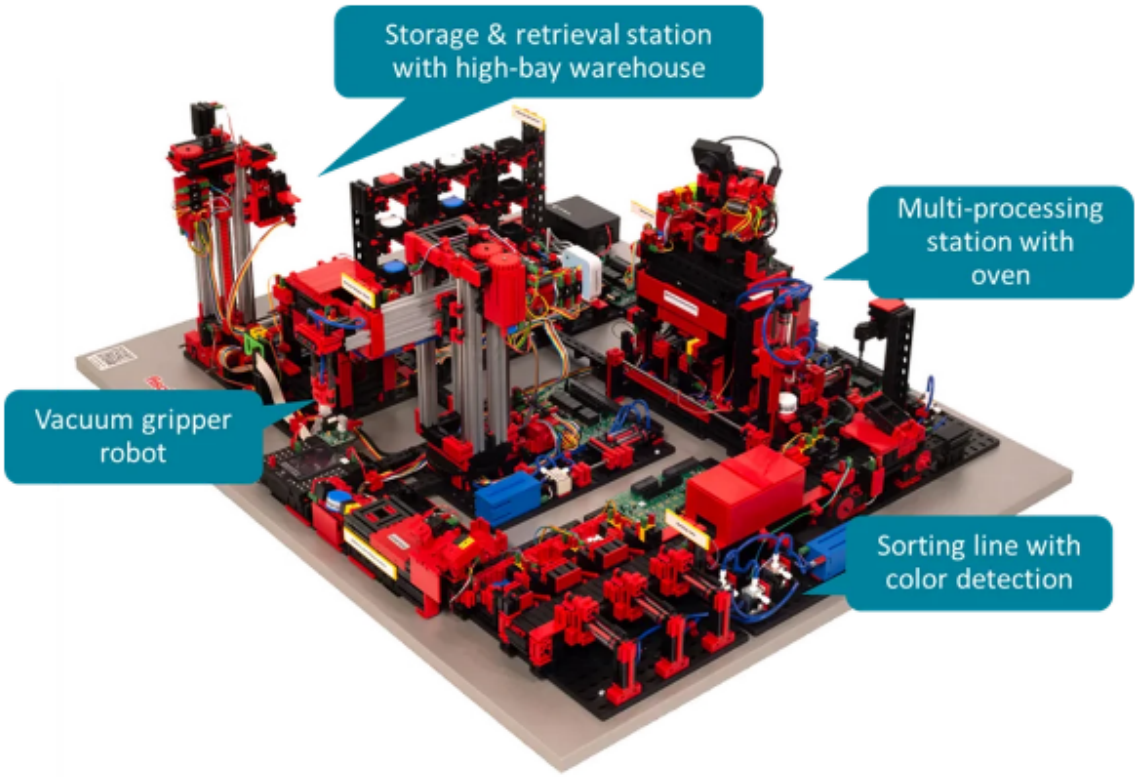

Figure 1: TNO-ESI demo system.

Figure 2: The digital twin.

Each subsystem performs a specific function within this workflow, with products moving sequentially from the warehouse to the multi-processing station and finally the sorting line and back to the warehouse. Figure 3 illustrates the entire product flow through the system.

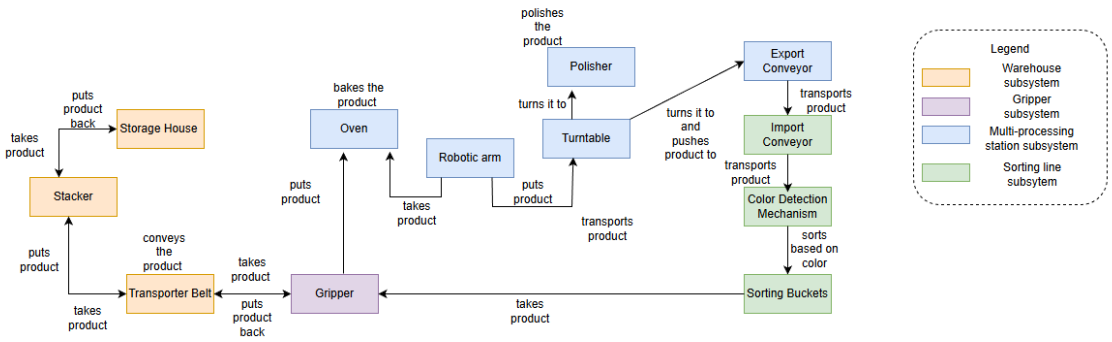

Figure 3: Product workflow through the TNO-ESI demo system.

The first subsystem is the **warehouse**, as shown in Figure 4, which serves as the initial stage of the production process. This subsystem includes a storage unit, a stacker, and a transport belt that work together to handle product flow. The storage unit contains nine different products, each distinguished by color and initially placed within designated buckets. The initial configuration of the storage house follows the Dutch flag pattern, where products are arranged by color in three rows: red on top, white in the middle and blue on the bottom. The stacker functions as a three-axis crane capable of motion along the x, y, and z directions. Although the stacker has access to the three-dimensional space,

the body can only move on the x and z axes. T he cantilever ensures the movement into the y plane. The cantilever is a structural arm that extends horizontally from a fixed support point. Its primary motion is horizontal extension and retraction. In addition, the cantilever supports a limited vertical lifting movement, which is used solely to lift a product off its support or to place it down. During a pickup operation, the stacker first moves the cantilever to a height just below the target product, after which the cantilever performs a small upward movement to lift the product. The primary task of the stacker is to retrieve products from the storage buckets and position them onto the transport belt. This transport belt transports the selected product from the warehouse to the gripper subsystem and subsequently moves back to its initial position. In doing so, it serves as the interface between the warehouse and the subsequent subsystems.

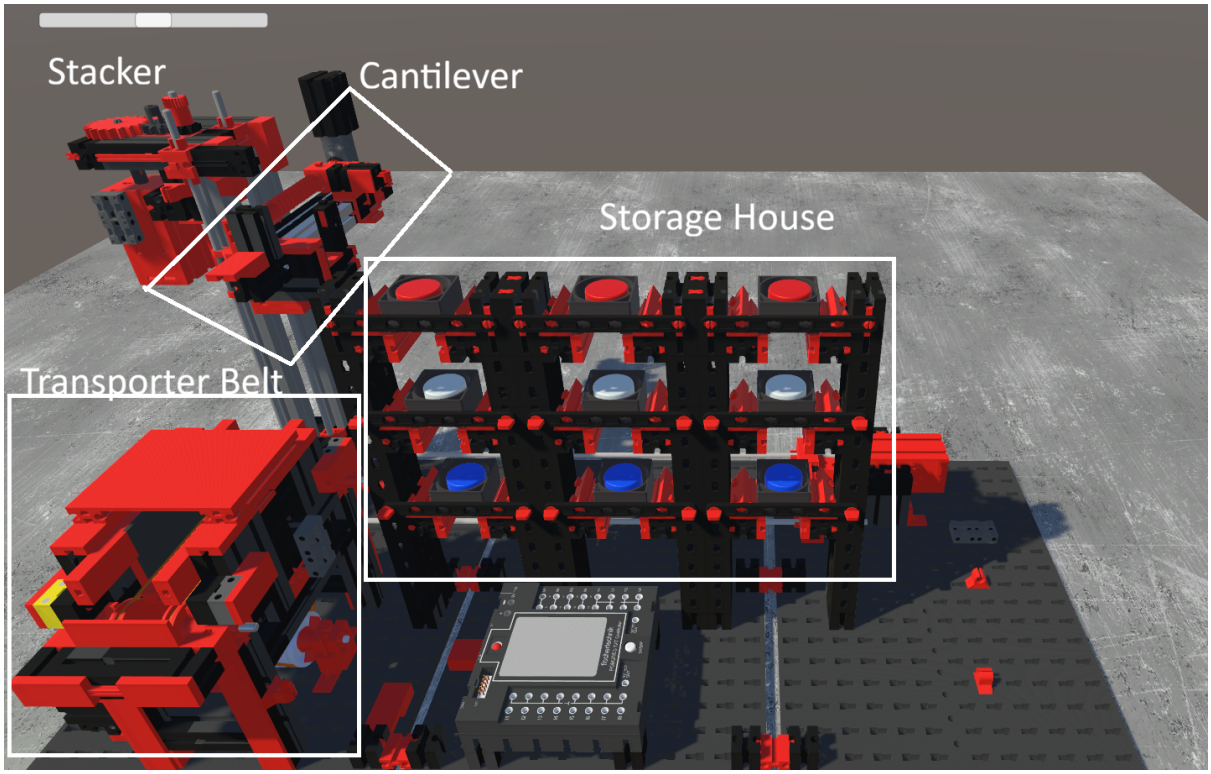

Figure 4: Warehouse subsystem.

The **gripper** (shown in Figure 5) operates as a crane-like mechanism that can move along the x and z axes, and can rotate 360 degrees. It employs a vacuum-based gripping mechanism to handle products. The gripper's purpose is to pick up products from within the system and place them at required locations, ensuring the correct handover of product between the warehouse, multi-processing station and sorting line subsystems.

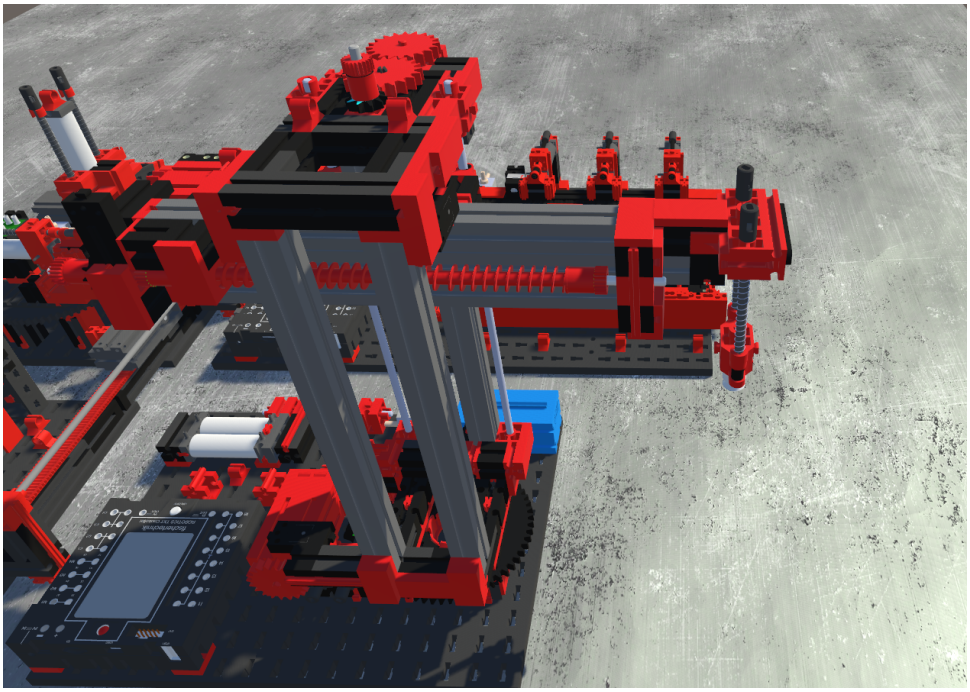

Figure 5: Gripper subsystem.

The third subsystem shown in Figure 6, the **multi-processing station**, represents the core of the manufacturing process and contains the largest number of components. Its central element is the oven, in which each product can undergo a mock "baking" process for a predefined time interval. Once processed, the product is transferred within the station using a robotic arm. This arm uses a vacuum-based gripping mechanism, similar to the gripper, but it is capable of moving only in the x plane. The station also includes a turntable, which rotates the product between three distinct positions: the receiving position, the polishing position, and the export position. In the polishing position, the product undergoes a mock "surface finishing" procedure designed only for demonstration. Once polished, the product is moved to the export position, where it is prepared for transfer to the sorting line.

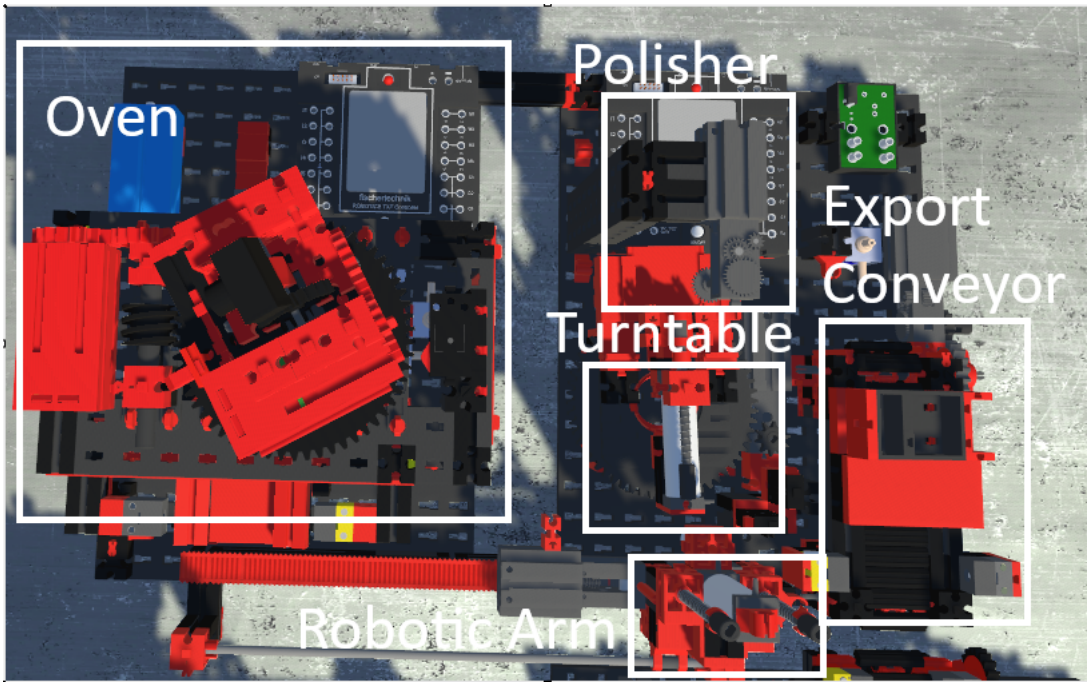

Figure 6: Multi-processing station.

The last subsystem is the **sorting line**, presented in Figure 7. It is responsible for classifying finished products based on their color. A color detection unit scans each incoming product and identifies its color. This information can subsequently be used to select the product's destined sorting location. From here, the gripper can move the product back to the transporter belt, which conveys it back to the stacker, until the product is back in the warehouse.

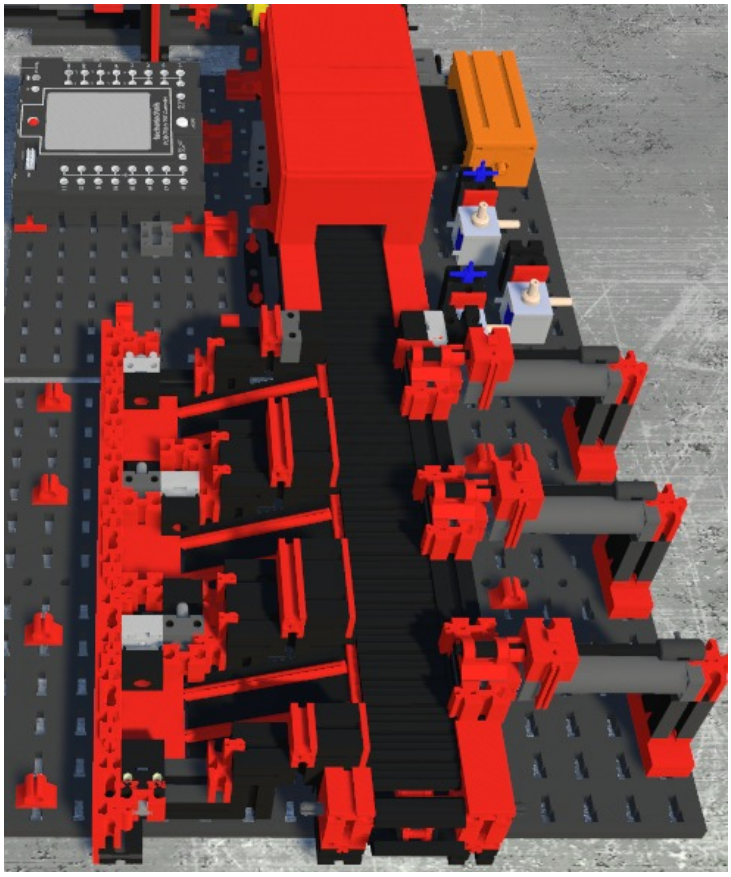

Figure 7: Sorting Line.

The four subsystems combine to create a modular system that can be configured to run different scenarios. Before any actions in the system can take place, each component must be homed, meaning that they need to be moved to a predefined starting position. Although we model each subsystem independently, they must work together during actual operation.

4.1 Digital Twin

Mirroring the physical system, the **Digital Twin** is the perfect tool to test the supervisory controller after the simulation step, before deployment on the actual hardware. This software is created in Unity [19] by the ICT Group. It mimics the behavior shown by the actual demo platform. This means that we can verify our CIF models and synthesized controllers without requiring constant access to the physical demonstration platform or without the concern of deteriorating the actual hardware of the system through wear and tear.

4.2 Communications

The operation of the TNO-ESI platform is coordinated by a Programmable Logic Controller (PLC), which executes control commands and manages real-time interactions among the system's physical components. Communication between the software layer and the hardware is established through the Open Platform Communications Unified Architecture (OPC UA), a standard protocol for industrial data exchange. To further

generalize this communication, the system utilizes an Industrial Hardware Abstraction Layer (IHAL). This middleware layer acts as a translation layer between the generated supervisory logic and the physical platform. This architecture is shown in Figure 8.

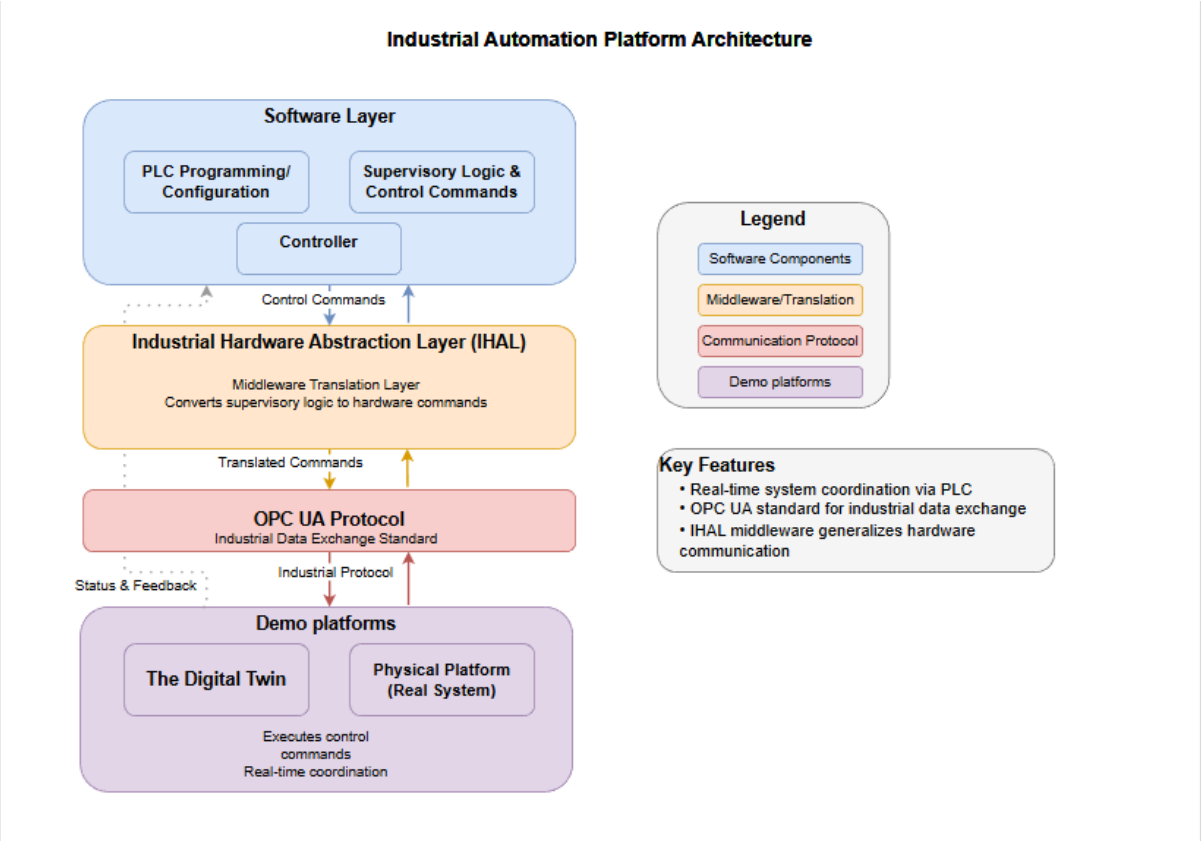

Figure 8: Architecture of the system.

4.2.1 IHAL interface

The IHAL interface defines a uniform command execution model for all system components, enabling the supervisory controller to coordinate multiple operations consistently. A fundamental characteristic of the IHAL design is the dual-node architecture: every command is represented by two distinct OPC UA nodes: an action node that initiates the operation and a completion node that signals when the operation has finished. This structure allows the controller to distinguish between command initiation and completion, which is essential for reliable coordination.

The IHAL also provides feedback mechanisms that allow the control system to monitor command execution status and respond appropriately to both successful completions and error conditions. Each command follows a request–response pattern, and the outcome of its execution is communicated through specific return codes:

- **Return code 500**: command successfully executed.
- **Return code 501**: command issued before the system has completed homing (a procedure that ensures that all the system components are in known locations, as discussed later in Section 5.7).
- **Return code 502**: physically impossible command issued.

- **Return code 503**: physical collision detected between system components.

5 Developing a formal CIF model of the system

One of the objectives of this project is to develop a formal model using CIF, that enables the automation of the TNO-ESI demonstration system. By modeling the system in CIF, the components of the demo system can be described in a structured way, thus facilitating the synthesis of a supervisory controller. This approach ensures that the resulting control logic is both correct-by-construction and can be adapted to various demonstrations that we will develop later.

5.1 CIF modeling

In order to formally model the four subsystems of the demo system using CIF, a plant automaton is developed. It represents the operational behavior of commands as defined by the IHAL interface. This modeling strategy is chosen deliberately, in order to ensure that the automatically generated code could be integrated with the IHAL middleware layer, therefore maintaining consistency between the supervisory control logic and the underlying hardware abstraction.

5.2 Command automaton

The core of the CIF models is represented by the plant automaton definition *Command*, as shown in Listing 1. This parameterized definition takes a boolean algebraic variable $fail\_condition$ as input, which enables modeling of failure scenarios. It is organized around two locations: *Idle* and *Executing*. Its behavior is governed by three events that explicitly reflect the execution semantics of commands defined in the IHAL interface. The controllable event c_start initiates the execution of a command and transitions the automaton from *Idle* to *Executing*. The uncontrollable event $u\_succeeded$ represents the successful execution of the command and corresponds directly to the success status returned by the IHAL interface. Conversely, the uncontrollable event u_failed models unsuccessful command execution and is triggered when the IHAL interface reports an error condition. In the case of a failure, the automaton returns to the *Idle* location.

In addition to events, the *Command* automaton introduces the input variable *executed*. This variable represents whether the physical or simulated system has completed execution of the command. Its value is not determined by the CIF model itself but is supplied by the execution environment via the IHAL interface. The automaton uses *executed* as a guard on the success transition, ensuring that successful completion can only occur once execution has actually finished in the underlying system.

To reflect practical operational constraints, the model assumes that at most one command per component may be executed at a time. This restriction is enforced at the system level by preventing multiple command automata from being in the *Executing* location simultaneously. As a result, no command can enter or complete execution while another command is already in progress.

```
1 plant def Command(alg bool fail_condition):
2
3     input bool executed;
4     controllable c_start;
5     uncontrollable u_failed, u_succeeded;
6
```

```
7       location Idle:
8           initial;
9           marked;
10          edge c_start goto Executing;
11
12      location Executing:
13          edge u_succeeded
14              when not fail_condition and executed = true
15              goto Idle;
16
17          edge u_failed
18              when fail_condition and executed = false;
19              goto Idle;
20
21  end
```

Listing 1: Command automaton.

After the creation of the generic *Command* automaton definition, we instantiate each command exposed by the IHAL interface. An example of this is shown in Listing 2, where we present how the commands used for retracting and extending the cantilever are instantiated. Furthermore a series of component-specific plant automata are developed to represent the operational behavior of each element within the warehouse, gripper, multi-processing station and sorting line subsystems. It is important to note that the command automata only capture whether commands are idle or in progress as they do not represent the component states reached after executing those commands. Therefore, to capture the actual states and interactions of each component through commands, we model each physical component separately with its own plant automaton.

```
22  extendCanteliver: Command(retractCantilever.Executing);
23  retractCanteliver: Command(extendCantilever.Executing);
```

Listing 2: Instantiations of the cantilever commands.

5.3 Warehouse

As described in the previous section, the warehouse contains three different components: the stacker, the transporter belt and the storage house. We start by first expressing how the stacker is modeled in CIF, as it represents the connecting element between the other two components of this subsystem.

5.3.1 Stacker Automaton

The *Stacker* automaton models the three-dimensional movement and product handling capabilities of the warehouse crane system. A total of eleven movements and their interactions are captured within a single integrated automaton that coordinates the stacker's complete operational behavior. These commands represent the discrete positions that the stacker can reach and they mirror the access patterns used by the stacker to retrieve products at different locations. At the modeling level, these commands abstract away concrete spatial coordinates. After code generation, they are subsequently mapped to

the actual stacker coordinates of the system through the interface layer that connects the generated controller to IHAL.

The movement along the x and z axes is represented by the discrete variable *move_pos*, which can take values such as *Idle*, *Belt*, *Storage1A*, *Storage1B*, *Storage1C*, *MovingTo*, or *UnknownMove*. The *Idle* state indicates that the stacker is at its home position. The specific storage positions like *Storage1A* denote the stacker's alignment with particular storage slots. The *Belt* value indicates alignment with the conveyor belt. The *MovingTo* denotes that a movement command is currently being executed, and *UnknownMove* represents an abnormal condition resulting from a failed movement command. To track movement trajectories, two auxiliary variables are maintained: $move\_from$ records the position from which movement originated, and *move_to* indicates the target destination. The transitions between positions are triggered by command starting events such as $moveToBelt.c\_start$ or $moveToStorage1A.c\_start$, with successful completion marked by the corresponding *u_succeeded* events.

The vertical movement of the cantilever is captured by the discrete variable *up_down_pos*, which can take one of four values: *MovingUpDown*, *Up*, *Down*, or *UnknownUpDown*. The *Down* state indicates that the cantilever is in its lower position, whereas *Up* denotes the raised position used for retrieving products from storage. The *MovingUpDown* value indicates that a vertical movement command is currently being executed and *UnknownUpDown* represents an abnormal condition resulting from a failed command. Similar to horizontal movement tracking, the variables up_down_from and *up_down_to* record the source and destination positions for vertical movements. For instance, executing the $stackerToUpPos.c\_start$ command moves the cantilever into the *MovingUpDown* value, transitioning to *Up* when the *u_succeeded* event is triggered. Similarly, the $stackerToDownPos.c\_start$ command returns the cantilever to its *Down* position.

The movement along the y axis of the cantilever is represented by the discrete variable *in_out_pos*, which controls the cantilever extension and retraction. This variable can take values: *Moving*, *In*, *Out*, or *Unknown*. The *In* state indicates that the cantilever is retracted, whereas *Out* denotes that it is extended to reach products. The *Moving* value indicates that an extension or retraction command is currently being executed and *Unknown* represents an abnormal condition. The position tracking variables in_out_from and *in_out_to* maintain the movement trajectory. Executing $extendCantilever.c\_start$ moves the cantilever into the *Moving* state, transitioning to *Out* once the *u_succeeded* event is triggered. Similarly, retracting the cantilever follows the reverse process by executing $retractCantilever.c\_start$, which transitions the cantilever from *Out* back to *In*.

In addition to movement coordination, the *Stacker* automaton also models product handling and transfer logic through synchronized events with the *StorageHouse* and *Transporter* automata. The product pick-up sequence involves a coordinated movement pattern. First, the cantilever extends outward (y-axis movement to *Out* position) by executing $extendCantilever.c\_start$. After that, it performs a small upward lift to engage with the product by executing $stackerToUpPos.c\_start$ and finally retracts inward. During this retraction, the $retractCantilever.u\_succeeded$ event is synchronized with the corresponding event in the storage unit. This enables the stacker to obtain the product only if it is in the correct positions (*Out* and *Up*) and the storage slot contains a product. Similarly, when the *Stacker* places a product, it extends the cantilever, lowers to the *Down* position to release the product, and then retracts. When placing products onto the belt, synchronization between $retractCantilever.u\_succeeded$ and the *Transporter's*

holds_in variable ensures that the product is successfully transferred. These synchronized edges guarantee atomic handovers, preventing partial or inconsistent product states across components. In Listing 3 we have a simplified pseudocode snippet that exemplifies the operational actions of the *Stacker* automaton.

```
1  plant  Stacker:
2      enum Move = Idle, Belt, Storage1A, Storage1B, Storage1C,
       Storage2A, Storage2B, Storage2C, Storage3A, Storage3B,
       Storage3C, MovingTo, UnknownMove;
3      enum MoveFromTo = NotMovingTo, IdlePos, BeltPos, StoragePos1A
       , StoragePos1B, StoragePos1C, StoragePos2A, StoragePos2B,
       StoragePos2C, StoragePos3A, StoragePos3B, StoragePos3C,
       UnknownMovePos;
4      disc Move move_pos = Idle;
5      disc MoveFromTo move_to = NotMovingTo;
6      disc MoveFromTo move_from = NotMovingTo;
7      \\similar enums for
8      ...
9
10     disc Product holds = Nothing;
11      location:
12         initial;
13         marked;
14         //homing mechanism
15     ...
```

Listing 3: Pseudocode of the Stacker automaton.

5.3.2 Transporter Automaton

The *Transporter* automaton models the behavior of the conveyor belt that connects the warehouse's stacker to the gripper subsystem and serves as an intermediate buffer for product transfer, as can be seen in Listing 4. Its primary responsibility is to capture both the physical motion of the belt and the precise handover of products between adjacent subsystems.

The operational status of the *Transporter* automaton is represented by the discrete variable *transporter_state*, which can take one of three values: *Idle, Moving, or Unknown*. The *Idle* state indicates that the belt is stationary and available for interaction, *Moving* denotes that a transport command is currently being executed, and *Unknown* represents an abnormal condition resulting from a failed command or an unsuccessful homing operation.

To model product flow, the automaton maintains two discrete variables: *holds_in* and *holds_out*. These variables track the presence of a product on the stacker-facing side and the gripper-facing side of the belt, respectively, as presented in Figure 9. This dual-buffer abstraction allows the model to distinguish between incoming and outgoing product positions and to enforce directional constraints on product movement.

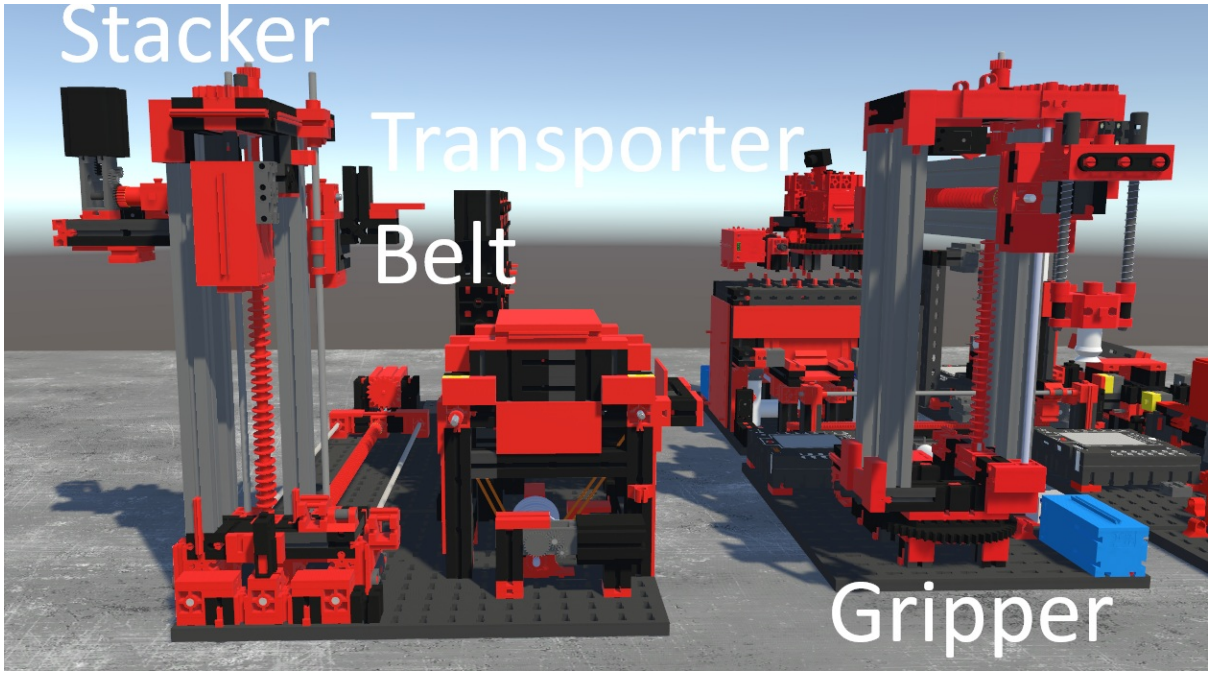

Figure 9: Transporting belt flow.

The *Transporter* is initialized in the *Idle* state and participates in the global homing procedure of the warehouse.

Product transportation along the *Transporter* is modeled through two command automata: *transportBoxToGripper* and *transportBoxToStacker*. When either command is initiated, the belt enters the *Moving* state. After the successful execution of *transportBoxToGripper*, a product located at the *Stacker* side (*holds_in*) is transferred to the gripper side (*holds_out*). Conversely, successful execution of *transportBoxToStacker* transfers a product from *holds_out* back to *holds_in*. In both cases, failures result in a transition to the *Unknown* state, preventing further operations until the system is recovered.

In addition to the motion, the *Transporter* automaton plays a central role in coordinating product handovers with both the *Stacker* and the *GripperRobot* through synchronized events. When the *Stacker* lowers a product onto the *Transporter* and retracts the cantilever (*retractCantilever.u_succeeded*), the transition is synchronized such that *holds_in* is updated with the product carried by the *Stacker*. This can only happen provided that the *Transporter* position is free, the *Stacker* is positioned at the *Belt* position, and the cantilever is extended and lastly retracted. Similarly, when the *Stacker* lifts a product from the *Transporter* and then retracts itself, the product is removed from *holds_in* if all positional and availability conditions are satisfied.

```
plant Transporter:
    enum TransporterState = Moving, Idle, Unknown;
    disc TransporterState transporter_state = Idle;
    disc Product holds_in = Nothing;  // Product closer to
    stacker side
    disc Product holds_out = Nothing; // Product closer to
    gripper side

    location:
        initial;
        marked;

        // Homing mechanism
        edge homeWarehouse.c_start do transporter_state := Moving
        ;
        edge homeWarehouse.u_succeeded do transporter_state :=
        Idle;
        edge homeWarehouse.u_failed do transporter_state :=
        Unknown;

        //  motion: to gripper
        edge transportBoxToGripper.c_start do transporter_state
        := Moving;
        edge transportBoxToGripper.u_succeeded do
            transporter_state := Idle,
            holds_out := holds_in,
            holds_in := Nothing;
        edge transportBoxToGripper.u_failed do transporter_state
        := Unknown;

        //  motion: to stacker(same as the motion to the gripper
        but use transportBoxToStacker command)
        ...

        // Handover with Stacker
        edge retractCanteliver.u_succeeded do
            // Stacker takes product from belt
            if holds_in != Nothing and Stacker.holds = Nothing
               and Stacker.move_pos = Stacker.Belt and Stacker.
               up_down_pos = Stacker.Up:
                holds_in := Nothing
            // Stacker places product on belt
            elif holds_in = Nothing and Stacker.holds != Nothing
                 and Stacker.move_pos = Stacker.Belt and Stacker.
                 up_down_pos = Stacker.Down:
                holds_in := Stacker.holds

            end;
...
end
```

Listing 4: Transporter automaton.

5.3.3 StorageHouse Automaton

The storage unit is modeled using nine separate *StorageHouse* automaton instances, one for each physical storage slot in the warehouse, as shown in Listing 5. Each storage slot can hold a product of a specific color ('Red', 'White', or 'Blue') defined through the *Product* enumeration. Every *StorageHouse* instance is parameterized with a storage position identifier (*move*) and an initial product configuration (*initialProd*), allowing each slot to be initialized with different contents. Each *StorageHouse* instance tracks the presence or absence of a product in its corresponding slot via its own *holds* variable and updates it in response to synchronized hand-over events with the stacker. The automaton contains a single location with an edge synchronized on the *retractCantilever.u_succeeded* event that implements the product transfer logic. For product retrieval, when the *Stacker* executes the *retractCantilever.u_succeeded* event while positioned at a particular storage location (*Stacker.move_pos* = *move*) in the *Up* position, the *Stacker* is not already holding a product (*Stacker.holds* = *Nothing*), and that slot's product is available (*holds != Nothing*), the corresponding *StorageHouse* instance sets its variable *holds := Nothing*, representing that the item has been successfully retrieved from that specific slot. Conversely, for product placement, when the *Stacker* is in the *Down* position at the storage location and the same retracting command is executed, if the *Stacker* carries a product (*Stacker.holds != Nothing*) and the storage location is empty (*holds* = *Nothing*), the automaton updates the state to *holds := Stacker.holds*, thereby reflecting a successful placement operation and transferring the product from the stacker into the storage slot.

```
1  plant def StorageHouse(alg Stacker.Move move; alg Product
   initialProd):
2      // Variable for tracking the state of the product within the
3      // Storage automaton
4      disc Product holds = initialProd;
5
6      location:
7          initial;
8          marked;
9
10         // Product handover within the StorageHouse automaton
11         edge retractCantilever.u_succeeded do
12             if holds != Nothing
13                and Stacker.holds = Nothing
14                and Stacker.move_pos = move
15                and Stacker.up_down_pos = Stacker.Up:
16                 holds := Nothing
17             elif holds = Nothing
18                  and Stacker.holds != Nothing
19                  and Stacker.move_pos = move
20                  and Stacker.up_down_pos = Stacker.Down:
21                 holds := Stacker.holds
22             end;
23 end
```

Listing 5: StorageHouse automaton definition.

5.4 GripperRobot Automaton

The gripper subsystem is modeled as a plant automaton representing the crane responsible for transporting products between the warehouse, the multi-processing station, and the sorting line. Its behavior is governed by two principal types of motion: (1) positional movement, defining the horizontal and vertical displacements of the gripper between locations (e.g., *Belt*, *Oven*, *Sort1*, *Idle*), and (2) gripping actions, specifying the act of grasping or releasing a product using a vacuum-based mechanism. Edges are explicitly defined to handle both successful and failed executions, ensuring that the modeled behavior remains consistent with the real-world system's response to communication errors or mechanical faults.

To track the gripper's movement trajectory, the automaton *GripperRobot* maintains two state variables: *move_to*, which stores the destination location where the gripper is moving or intends to move, and *move_from*, which records the origin location from which the gripper started its current movement. These variables work together with the current position indicator to provide complete information about the gripper's state during transitions, enabling proper synchronization with other subsystems and supporting error recovery by maintaining a record of both the source and target positions throughout each movement operation.

The *GripperRobot* automaton also includes a product-tracking variable, called *holds*, which records whether it currently carries a product. The value of this variable changes as a result of synchronized handover events with other subsystems. For instance, when the *GripperRobot* executes an *armGrab.u_succeeded* transition while having the positional value of *Belt*, synchronization occurs with the *Transporter* automaton's *u_succeeded* transition, resulting in the product being transferred from the *Transporter's* output position to the *GripperRobot's* holding state. Similarly, when the *GripperRobot* releases a product at the *Oven* location or the *Belt* value (*armRelease.u_succeeded* at *move_pos* = *Oven* or *move_pos* = *Belt*), synchronization with either the *BakingOven* or *Transporter* automaton ensures that the product is deposited correctly onto the oven tray or the belt. The handover of the products can be seen in Listing 6.

```
1 edge armGrab.u_succeeded do
2     if move_pos = Oven
3         and BakingOven.tray_state_pos = BakingOven.Out
4         and BakingOven.holds != Nothing
5         and holds = Nothing:
6          holds := BakingOven.holds
7     end;
```

Listing 6: Handover of the *GripperRobot* with the *BakingOven*.

5.5 Multi-processing station

The multi-processing station is modeled in a similar manner. This subsystem consists of four elements, each of them being modeled as plant automata: *BakingOven*, *RoboticArm*, *TurntableMover* and *ExportConveyor*. the desired product flow is that first a product enters the *BakingOven*, being then transported to the *TurntableMover* by using the *RoboticArm* and lastly the element is pushed to the *ExportConveyor*. The *BakingOven* automaton captures the behavior of product processing operations, such as baking, rep-

resented through state transitions that define the product's position (*In, Out*, or *Moving*) and tray status. The *RoboticArm*, in turn, is modeled analogously to the *GripperRobot*, featuring the same vacuum-based mechanism for transferring products between the *Oven* and the *TurntableMover*. This uniformity in modeling structure allows for consistent interaction patterns and simplifies synchronization across the two stations.

Within the *TurntableMover* automaton, the movement of a product between the *Receive*, *Polish*, and *Export* positions is done by using commands that rotate the turntable. Each rotation command is represented by an instance of the *Command* automaton definition, while product handovers with the *RoboticArm* automaton are again managed through synchronized events. For example, when the robotic arm places a product at the *Receive* position *(armRelease.u_succeeded)*, this event is synchronized with the *TurntableMover's* corresponding *u_succeeded* event to ensure that the product state is updated simultaneously in both components as can be seen in Listing 7.

```
1 edge ArmRelease.u_succeeded do
2     if holds = Nothing
3         and OvenArm.holds != Nothing
4         and OvenArm.move_pos = OvenArm.Turntable
5         and move_pos = Receive:
6          holds := OvenArm.holds
7     end;
```

Listing 7: Handover between *TurntableMover* and *RoboticArm*.

5.6 Sorting line

The final element of the demo system is the sorting line. This subsystem is modeled using the same principles as the preceding components, with explicit emphasis on command-based execution and synchronized event handling between the *Gripper*, the *ImportConveyor* and the *ExportConveyor*. The sorting line automaton represents a color-based sorting mechanism, using an input variable *detect_color*. At the core of this subsystem lies the *ColorDetection* automaton, which models the color detection mechanism. Products are first transferred from the *ImportConveyor* to the *ColorDetection* station through synchronization on the event *conveyorImport.u_succeeded*. Upon reception, the product is stored in the discrete variable *holds* and the automaton transitions to the *HasProduct* state. The actual color detection process is initiated through the controllable event *detectColor.c_start* (shown in Listing 8), which moves the automaton into the *Scanning* state. Completion of the sensing operation is modeled by the uncontrollable event *detectColor.u_succeeded*.

```
edge detectColor.u_succeeded do
    if state = Scanning and holds != Nothing:
        if detected_color != Nothing:
            state := ReadyToSort
        else
            state := HasProduct   // Failed detection, retry
            possible
        end
    end;
```

Listing 8: Color detection mechanism.

The detected color itself is provided to the CIF model via the input variable *detected_color*, which represents the measurement from the physical color sensor. This input variable is not controlled by the CIF model but is instead supplied by the execution environment, interfacing directly with the actual color sensor hardware in the physical system or its simulated counterpart in the digital twin. The value of *detected_color* is updated by the execution environment to reflect sensor feedback from the physical system. If a valid color is detected, the automaton transitions to the *ReadyToSort* state, otherwise, the product remains available for a repeated detection attempt.

The sorting action is initiated via the controllable event *sortToBucket.c_start*, moving the automaton into the *MovingToBucket* state. Upon successful completion, indicated by *sortToBucket.u_succeeded*, the product is then transferred to the corresponding *SortingBucket* automaton, provided that capacity constraints are satisfied. Once the product has been deposited, the *ColorDetection* automaton returns to the *Idle* state, making the sorting line available for the next product. Products stored in the sorting buckets can subsequently be retrieved by the *GripperRobot* through synchronized grab events, enabling further transport within the factory.

It should be noted that the sorting logic presented in Listing 9 is implemented within the plant model itself, which represents a modeling design choice made to maintain simplicity in the current system architecture. Conceptually, sorting decisions determining which bucket a product should be routed to based on its detected properties is typically the responsibility of a supervisory controller rather than the plant model, as the plant should ideally only represent the physical system's capabilities and state, not the decision-making logic. However, for the demonstrations and use cases considered in this work, implementing this simple sorting rule directly in the plant model is sufficient and avoids the additional complexity of a separate decision-making layer. This approach is sufficient for the current scope but can be refactored in future work, such that we can employ more sophisticated sorting strategies.

```
edge sortToBucket.u_succeeded do
    if state = ProductMovingToBucket and holds != Nothing:
        // Check if target bucket has space
        if detected_color = Red and sortingBucket1.hasSpace:
            holds := Nothing
            state := Idle
        elif detected_color = White and sortingBucket2.hasSpace:
            holds := Nothing
            state := Idle
        elif detected_color = Blue and sortingBucket3.hasSpace:
            holds := Nothing
            state := Idle
        else
            state := ProductMovingToBucket
    end;
```

Listing 9: Sorting mechanism based on color detection.

5.7 Homing procedure

Before any operational behavior is allowed, the elements of the subsystems must be homed. This is coordinated through four different automata: *HomeWarehouse*, *HomeGripperRobot*, *HomeMultiProcessing* and *HomeSortingLine*, which together manage the homing life cycle for all components. Homing is a critical prerequisite required by the IHAL interface for correct system behavior, as it ensures that all mechanical components start from known reference positions and that subsequent motion commands can be executed safely.

The automata maintain two discrete boolean variables to capture the homing status of the system. The variable *isHomed* indicates whether a successful homing operation has been completed since the start of the system or the last command failure. This variable is used by system requirements to permit or restrict the execution of other commands. The variable *homingInProgress* reflects whether a homing operation is currently being executed, preventing concurrent homing requests and enabling consistent synchronization with other components.

Initially, the system is in an not homed state, with *isHomed* set to false. When the homing process is initiated via the controllable event *c_start*, the automaton marks that a homing operation is in progress by setting *homingInProgress* to true. This transition represents the start of a coordinated homing sequence across all warehouse components, as can be seen in Listing 10.

```
plant HomeWarehouse:
    // True if a successful homing has completed since the last
    failure or system start
    disc bool isHomed = false;
    disc bool homingInProgress = false;

    location:
        initial;
        marked;

        // Homing lifecycle
        edge homeWarehouse.c_start do
            homingInProgress := true;

        edge homeWarehouse.u_succeeded do
            homingInProgress := false,
            isHomed := true;

        edge homeWarehouse.u_failed do
            homingInProgress := false,
            isHomed := false;
end
```

Listing 10: HomeWarehouse automaton.

Upon successful completion of the homing operation, indicated by the uncontrollable event *u_succeeded*, the automaton updates its state to reflect that the system has been successfully homed. The variable *isHomed* is set to true and the variable *homingInProgress* is cleared. At this point, the system is considered to be in a safe initial configuration, and other operational commands may be enabled by the supervisory controller.

6 Simulation model

To validate the correctness of the developed CIF models and to ensure that the specified supervisory behavior aligns with the intended physical operation of the TNO-ESI demonstration system, a simulation and visualization layer is implemented. This layer enables both logical and visual validation of system dynamics by mapping internal CIF variables to graphical elements that represent the physical components of the factory platform.

6.1 TimedMove Automaton

To capture realistic temporal behavior, the simulation model incorporates timed automaton patterns using an automaton definition named *TimedMove*. This automaton encapsulates the execution semantics of time-consuming actions by associating a command with a predefined execution duration. Each instance of *TimedMove* models a single command (e.g., *retractCantilever, extendCantilever, armGrab*) as a two-location automaton consisting of an *Idle* and an *Executing* state, shown in Listing 11.

```
1  automaton def TimedMove(
2      controllable c_start;
3      uncontrollable u_succeeded;
4      uncontrollable u_failed;
5      alg real duration
6  ):
7      cont t0 = 0 der 1;
8      disc bool succeeded = false;
9
10     location Idle:
11         initial;
12         edge c_start do
13             t0 := 0.0,
14             succeeded := true
15         goto Executing;
16
17     location Executing:
18         edge u_succeeded
19             when t0 >= duration
20             do succeeded := true
21             goto Idle;
22         edge u_failed;
23 end
```

Listing 11: Timed movement automaton definition.

The temporal evolution of command execution is modeled using a continuous clock variable t_0, whose derivative is defined as t_0'=1. After the activation of the controllable start event *c_start*, the clock is reset and the automaton transitions to the executing location. Once the elapsed time satisfies the specified duration constraint, an uncontrollable completion event *u_succeeded* is triggered, returning the automaton to the *Idle* location. This construction enables the explicit modeling of execution delays and avoids instantaneous transitions, thereby producing smooth and physically plausible behavior

during simulation. This behavior is possible by instantiating time commands that define the physical components described in Section 4. For example the action of retracting or extending the cantilever is instantiated as a simulation variant using *TimedMove*, as shown in Listing 12.

```
1   extendCanteliverTiming : TimedMove(extendCanteliver.c_start,
    extendCanteliver.u_succeeded, extendCanteliver.u_failed, 5.0)
    ;
2   retractCanteliverTiming : TimedMove(retractCanteliver.c_start
    , retractCanteliver.u_succeeded, retractCanteliver.u_failed,
    5.0);
```

Listing 12: Instatiation of the ArmGrabTiming command.

These instantiations use the same event names (*c_start*, *u_succeeded*, *u_failed*) that appear in the corresponding discrete-event *Command* automaton used for controller synthesis, but now associate them with realistic execution durations. The duration parameter represents the time in seconds required for the physical action to complete. By passing the controllable and uncontrollable events of the *Command* automaton to the timed variant, the two models remain structurally aligned while serving different purposes. The *Command* automaton captures logical execution constraints for supervisor synthesis, while the timed variant models realistic temporal behavior for simulation.

The use of timed automata serves two important purposes. First, it improves the realism of the simulation by ensuring that actuator movements, transport operations, and processing steps unfold over time rather than occurring instantaneously. Second, it allows the simulation to be aligned closely with the semantics of the command plant automata used for controller synthesis, which operate on the same event-based start and completion signals. To facilitate integration with the plant automata, the completion status of each timed command is exposed through a variable that can be merged with the plant model's input variables. As mentioned in Section 5.2, each *Command* automaton uses an *executed* variable derived from the internal *succeeded* flag of the timed automaton. This variable is directly merged with the corresponding input variables of the plant model through a merge step performed before simulation. This merge converts an open model with undefined inputs into a closed model where everything is specified internally, linking the simulation model's discrete variables to the plant model's inputs and enabling the simulation to provide the values the plant needs. This ensures the simulation and synthesis models work together consistently.

As stated in Section 2.2, a simulation model can use urgent or non-urgent events. The simulation model's handling of event urgency depends on whether a synthesized controller is present. Without a synthesized controller, the simulation model ensures that time can progress continuously, even when discrete state transitions are enabled. This is made possible by using non-urgent events. Normally, all events in CIF are urgent by default, meaning that if an event is enabled, it must be taken immediately and time cannot progress. Non-urgent events change this behavior by allowing time to advance even when the event is enabled. In the manual simulation model, the uncontrollable completion events (*u_succeeded*, *u_failed*) are made non-urgent. This allows the continuous clock variable t_0 to progress and reach the specified duration threshold before the completion event is actually taken, thereby modeling realistic physical movements and delays. For interactive SVG visualizations, clicking on graphical elements within the

SVG environment determines which uncontrollable events become active. In this interactive simulation mode, events linked to user clicks remain urgent, and their enabling is controlled by the user's interactions with the visualization rather than by non-urgency declarations. With a synthesized controller, all events should remain urgent. The temporal evolution occurs naturally through the use of the *TimedMoved* automaton definition. Once the time constraint is satisfied, the event becomes enabled and is taken immediately. This ensures proper synchronization between the controller and the physical system.

This design ensures that time-dependent behaviors such as the movement of products along conveyor belts, the extension of the cantilever, or the rotation of the gripper are accurately represented in the simulation with realistic durations.

6.2 SVG Based Visualization

The visualization environment is based on the Scalable Vector Graphics (SVG) format [17], which provides a flexible, resolution-independent, and high-fidelity representation of the demo system. The graphical representation of the factory is created using Inkscape [11], a vector-graphics editor well suited for designing complex SVG-based layouts. During the design process, each mechanical component of the factory such as the stacker, storage houses, gripper, conveyor belts, and robotic arms is modeled as an individual SVG element with a unique identifier. This means that every element is animated to match the mechanical characteristics of the physical 3D system. For example, the stacker is animated using vertical linear translations to represent its movement along the x axis, and the cantilever uses horizontal translation to show extension and retraction. The coordinate system used in the SVG visualization follows Inkscape's standard convention, where the origin $(0, 0)$ is located at the top-left corner of the canvas, the horizontal axis represents the x direction (increasing to the right), and the vertical axis represents the y-direction (increasing downward).

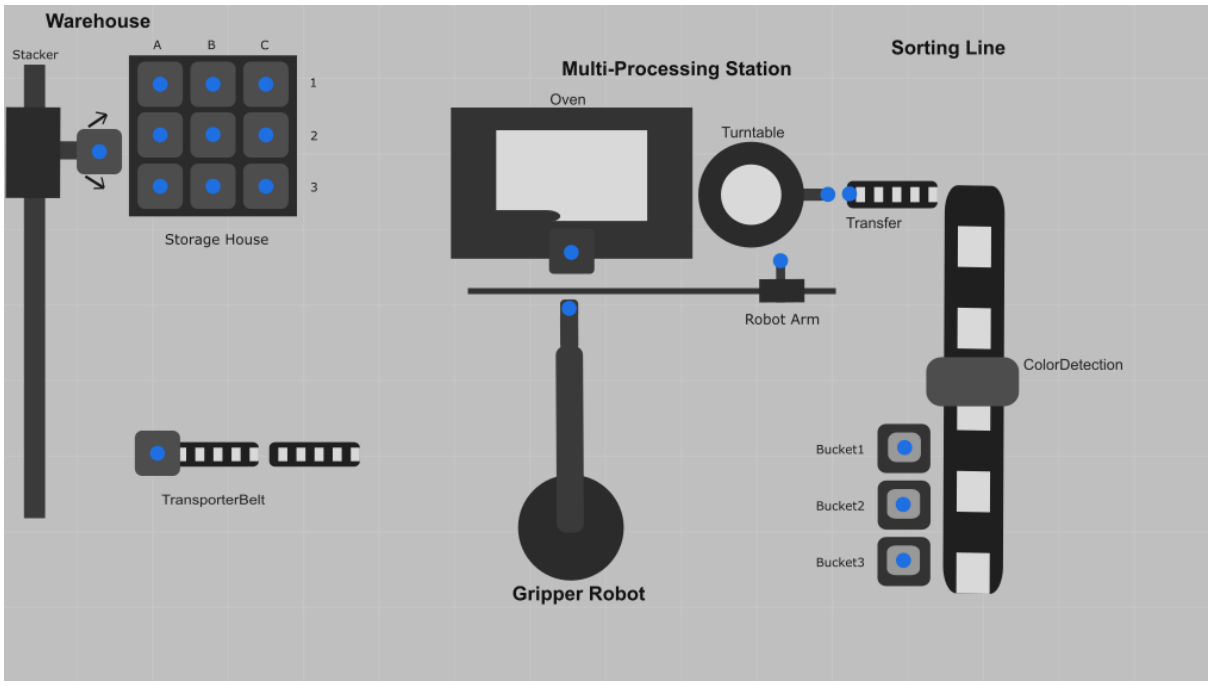

Figure 10: Simulation model.

CIF supports the use of SVG files to visualize system movements and component interactions during simulation. As shown in Figure 10, this is the SVG image used for the simulation. This visualization provides a graphical representation of the model's behavior, making it easier to understand and verify the system's operation. In the CIF code, the *scale* function is used to smoothly interpolate between positions during timed movements. The function takes five parameters: the current command time value (t_0), the command start time, the command duration, the start position coordinate, and the end position coordinate. It linearly interpolates between the start and end coordinates based on how much time has elapsed for the command.

The stacker moves vertically in SVG, which corresponds to movement along the physical x axis of the warehouse. The negative y coordinates in Inkscape represent positions further down the canvas, which correspond to different storage positions in the physical system. During simulation, *move_from* and *move_to* serve critical roles in monitoring, validating, and visualizing system behavior. They provide visibility into ongoing movements by capturing both the source and destination of each transition, which is essential for debugging movement sequences and ensuring that the stacker follows expected trajectories. These variables are particularly important for the visualization layer, where they enable smooth animated transitions of the stacker's position in the graphical representation. For example, when the stacker initiates movement from the *Belt* position to *Storage1A*, *move_from* is set to *BeltPos* and *move_to* is set to StoragePos1A, while *move_pos* transitions to *MovingTo*. The visualization system uses this information to interpolate the stacker's position between the source and destination coordinates over time, creating a smooth animation. This is implemented in the SVG output code, as shown in Listing 13.

```
1  svgout id "Stacker" attr "transform" value
2  fmt("translate(0,%s)",
3      if Stacker.move_pos = Stacker.MovingTo:
4          if Stacker.move_from = Stacker.UnknownMovePos
5             and Stacker.move_to = Stacker.StoragePos1A:
6              scale(moveToStorage1ATiming.t0, 0.0, 10.0, -86.0,
               -440.0)
7          elif Stacker.move_from = Stacker.BeltPos
8               and Stacker.move_to = Stacker.StoragePos1A:
9              scale(moveToStorage1ATiming.t0, 0.0, 10.0, -2.0,
               -440.0)
10     ...
11     end
12 );
```

Listing 13: Stacker animation mapping (partial SVG output mapping).

Another example of movement is the cantilever, which extends and retracts horizontally in SVG, corresponding to the physical y-axis. When the cantilever extends outwards, it moves from position 0.0 (retracted) to position 20.0 (extended), as seen in Listing 14. This horizontal translation allows the stacker to reach into storage positions to pick up or place products.

```
svgout id "Extender" attr "transform" value
fmt("translate(%s,0)",
    if Stacker.in_out_pos = Stacker.Moving:
        if Stacker.in_out_to = Stacker.OutPos:
            scale(extendCantileverTiming.t0, 0.0, 5.0, 0.0, 20.0)
        elif Stacker.in_out_to = Stacker.InPos:
            scale(retractCantileverTiming.t0, 0.0, 5.0, 20.0,
            0.0)
        else
            0.0
        end
    ...
    end
);
```

Listing 14: Cantilever extension animation (partial SVG output mapping).

The gripper is animated using rotational transformations rather than linear translations, reflecting its physical design as a rotating crane mechanism. Unlike the stacker which moves linearly, the gripper rotates around a fixed pivot point to reach different positions throughout the factory. The 'rotate' transformation in SVG takes three parameters: the rotation angle (in degrees), and the x and y coordinates of the pivot point around which the rotation occurs. The gripper rotates around this center point to reach different operational positions. The first parameter, the rotation angle, is interpolated using the scale function as the gripper moves. For example, when the gripper moves from the *Oven* position to the *Belt* position, the angle is smoothly interpolated from -1.45 to -80.5 over a duration of 6.0 seconds. This creates a smooth, arc-like motion as the gripper swings from one position to another. The pivot point for the gripper's rotation is defined by constant coordinates at 546.0 and 624.0 respectively (shown in Listing 15), ensuring that the gripper always rotates around the same fixed point regardless of which position it's moving to or from.

```
svgout id "GripperMechanism" attr "transform" value
fmt("rotate(%s,546.0,624.0)",
    if Gripper.move_pos = Gripper.MovingTo:
        if Gripper.move_from = Gripper.UnknownMovePos
           and Gripper.move_to = Gripper.BeltPos:
            scale(gMoveToBeltTiming.t0, 0.0, 6.0, -1.45, -80.5)
        elif Gripper.move_from = Gripper.BeltPos
             and Gripper.move_to = Gripper.BeltPos:
            scale(gMoveToBeltTiming.t0, 0.0, 6.0, -80.5, -80.5)
    end
    ...
);
```

Listing 15: Gripper rotation animation (partial SVG output mapping).

In addition to animating the mechanical movements of components, the simulation also visualizes the products and their transfers between subsystems, as seen in Listing 16. Each product is represented by two distinct SVG elements: a container (bucket

or tray) and a colored element inside it. This two-part representation allows the simulation to clearly show which component currently holds each product and to animate product handovers between different parts of the system. The colored element's appearance is controlled by modifying its *fill* attribute based on the product type stored in the corresponding automaton's holds variable. For example, the export conveyor's product visualization is defined as Listing 16.

```
1 svgout id "ExportElement" attr "fill"
2     value switch ExportConveyor.holds:
3         case Red : "red"
4         case White : "white"
5         case Blue : "blue"
6         else  "#b3b3b3"
7     end;
```

Listing 16: Product color animation.

In the end each animation is tailored to match the physical capabilities and movement patterns of its corresponding real-world component.

The SVG-based visualization layer serves as a powerful diagnostic and validation tool throughout the development process. The reason of implementing this simulation by visualization is to observe the animated execution of commands and product transfers, thereby helping us in identifying errors such as incorrect synchronization, missing constraints or unintended interactions. For instance, mismatches between logical product handover and visual placement immediately reveal inconsistencies between the plant model and the intended physical behavior. The simulation model is fully integrated within the CIF toolchain. When modifications are made to the plant models or command definitions that change the state variables or structure, the visualization mappings in the simulation model must be updated accordingly to reflect these changes. However, an important advantage of this architecture is that changes to requirements or supervisory logic that add restrictions without altering the underlying state representation do not require updates to the visualization. Since the synthesis process only constrains which behaviors are permitted rather than changing what states can be represented, the existing visualization mappings remain valid. This means that once the simulation visualization is established for the plant model, it continues to work correctly even as different supervisory controllers are synthesized from varying requirement specifications, requiring manual updates only when the fundamental plant structure itself changes.

7 Formal requirements

In supervisory control theory, requirements are formal specifications that define the desired behavior of the controlled system. They are typically expressed as logical formulae or automata that restrict states and/or sequences of events that are permissible. Ultimately, requirements serve as input to the synthesis process, enabling the automatic generation of a supervisor that enforces the user's specified behavior. The desired behavior of the demo system is defined through formal requirements. To ensure a clearer overview of how the demo system's behavior is restricted, the requirements are organized into two categories: *safety* requirements, expressed by logical formulae, and *behavioral* requirement, being expressed using requirement automata.

7.1 Safety Requirements

Safety requirements prevent unsafe or invalid system behavior by restricting the execution of commands under undesired conditions. We model the requirements using state invariants, which are logical conditions that must hold in every state. These invariants restrict the allowed system behavior by ruling out unsafe or undesired states. In this work we use state/event exclusion requirement invariants, which prevent certain events from occurring when certain conditions are not satisfied, making sure that actions that violate safety conditions are prohibited. These requirements reflect physical and operational constraints of the factory components and ensure that actions are only performed when the system is in a valid state. These safety requirements manage to formalize the essential physical constraints of the system thus ensuring that the synthesized logic respects the operational limitations of the hardware and preventing damaging the system.

A fundamental safety constraint applied throughout the system is that no operational command may be executed before the corresponding subsystem has been successfully homed. This constraint is enforced by dedicated requirement groups for each subsystem, including the warehouse, gripper robot, sorting line, and multi-processing station.

For instance, the *HomeWarehouseRequirements* group (shown in Listing 17) restricts warehouse related commands such as cantilever actuation, stacker movement, and transport operations until the warehouse homing procedure has completed successfully.

```
group HomeWarehouseRequirements:
    // Nothing except homeWarehouse is allowed before the system
    is homed
    requirement {
        extendCantilever.c_start,
        retractCantilever.c_start,
        transportBoxToGripper.c_start,
        transportBoxToStacker.c_start,
        stackerToUpPos.c_start,
        stackerToDownPos.c_start,
        moveToBelt.c_start,
        moveToIdle.c_start,
        moveToStorage1A.c_start,
        moveToStorage1B.c_start,
        moveToStorage1C.c_start,
        moveToStorage2B.c_start,
        moveToStorage2C.c_start,
        moveToStorage3A.c_start,
        moveToStorage3B.c_start,
        moveToStorage3C.c_start,
        moveToStorage2A.c_start
    }
    needs HomeWarehouse.isHomed;
end
```

Listing 17: Homing requirements.

Analogous constraints are defined for the gripper robot *HomeGripperRobotRequirements*, the sorting line *HomeSortingLineRequirements*, and the multi-processing station *HomeMultiProcessingRequirements*. Each group explicitly states that the execution of the listed commands is only permitted when the corresponding *isHomed* condition holds.

In addition to homing constraints, component-specific safety requirements are imposed to prevent mechanically unsafe behavior. The *StackerRequirements* group (shown in Listing 18), for example, ensures that horizontal movements of the stacker are only allowed when the cantilever is fully retracted. This requirement prevents collisions and enforces a safe mechanical configuration during stacker motion.

```
group StackerRequirements:
    // Horizontal movement only when cantilever IN
    requirement {
        moveToBelt.c_start,
        moveToIdle.c_start,
        moveToStorage1A.c_start,
        moveToStorage1B.c_start,
        moveToStorage1C.c_start,
        moveToStorage2B.c_start,
        moveToStorage2C.c_start,
        moveToStorage3A.c_start,
        moveToStorage3B.c_start,
        moveToStorage3C.c_start,
        moveToStorage2A.c_start
    }
    needs Stacker.in_out_pos = Stacker.In;
end
```

Listing 18: Stacker horizontal movement requirements.

7.2 Scenario Requirements

While safety requirements define what behaviors are forbidden, scenario requirements specify the admissible sequence of the system's operation in order to achieve a desired production task. There are multiple ways to demonstrate the synthesized supervisor's capabilities. One approach, described in this section, is to define explicit scenario requirements that prescribe a specific sequence of operations using requirement automata. These automata constrain the sequencing and coordination of commands without explicitly controlling the plant.

The explicit scenario is captured by the *ScenarioTracker* automaton, shown in Listing 19. This automaton encodes a production scenario as a sequence of discrete steps, represented by a bounded integer variable *currentStep*. Each transition in the automaton corresponds to the start of a command and is guarded by conditions that ensure the previous command has been successfully completed.

```
requirement ScenarioTracker:
    disc int[0..51] currentStep = 0;
    location:
        initial;
        marked currentStep = 51;

        edge homeWarehouse.c_start
            when currentStep = 0
            do currentStep := currentStep + 1;

        edge homeGripperRobot.c_start
            when currentStep = 1 and homeWarehouse.Idle
            do currentStep := currentStep + 1;

        edge homeMultiProcessing.c_start
            when currentStep = 2 and homeGripperRobot.Idle
            do currentStep := currentStep + 1;

        edge homeSortingLine.c_start
            when currentStep = 3 and homeMultiProcessing.Idle
            do currentStep := currentStep + 1;

        edge moveToStorage2A.c_start
            when (currentStep = 4 and homeSortingLine.Idle)
            do currentStep := currentStep + 1;

        edge extendCantilever.c_start
            when (currentStep = 5 and moveToStorage2A.Idle)
            do currentStep := currentStep + 1;

        edge stackerToUpPos.c_start
            when (currentStep = 6 and extendCantilever.Idle)
            do currentStep := currentStep + 1;
    ...
end
```

Listing 19: Snippet of the first 6 steps of the ScenarioTracker.

The scenario begins by enforcing a fixed homing sequence for all subsystems. Once homing is complete, the automaton constrains the execution of warehouse operations (retrieving products from storage), product transfers (moving products via the gripper), processing steps (baking and polishing in the multi-processing station), and sorting actions (placing products in the correct sorting buckets based on color). Commands are only enabled when the system is in the appropriate state (as indicated by the *currentStep* variable) and the previous command is in an idle state, meaning no other commands are currently executing. The automaton structure ensures the product workflow throughout the whole system by carefully ordering the step such that each command's enabling conditions are satisfied. The sequential structure of the *currentStep* variable, combined with guards that check component idle states, prevents circular waiting conditions and guarantees that the system can always make forward progress toward completing the

production scenario.

In addition to the scenario-based automaton, a *MovementRequirements* group, as shown in Listing 20 is defined to globally restrict commands that are not relevant for the selected scenario. For this particular scenario, the goal is to move a single product from the storage unit to the gripper via the transport belt. The *MovementRequirements* group disables unused movements and actions that are not needed for this specific task, thereby reducing the behavioral space and simplifying the synthesis problem.

```
1  group MovementRequirements:
2     requirement {
3          moveToStorage1A.c_start,
4          moveToStorage1B.c_start,
5          moveToStorage1C.c_start,
6          moveToStorage2B.c_start,
7          moveToStorage2C.c_start,
8          moveToStorage3A.c_start,
9          moveToStorage3B.c_start,
10         moveToStorage3C.c_start,
11         turntableToReceive.c_start,
12         gMoveToSort1.c_start,
13         gMoveToSort3.c_start
14    } needs false;
15 end
```

Listing 20: MovementRequirements group.

7.3 Marked predicate

In this section we discuss the use of an alternate approach to scenario requirements, that exploits the power of supervisory controller synthesis. This approach uses marking requirements that define only the desired goal states, allowing the synthesis algorithm to automatically determine all safe paths to reach them. In the marked predicate approach, certain system states are labeled as being final to indicate desired or target configurations. In this approach, rather than specifying a detailed sequence of operations, we specify only the initial states (products at their starting positions) and the desired final states (products at their swapped positions) as marked states. The synthesis algorithm then automatically computes a supervisor that drives the system from the initial state to a marked state while respecting all safety constraints. This approach is conceptually more elegant and demonstrates the strength of supervisory controler synthesis: it eliminates the need to manually design scenario automata for each production task. In this scenario, when swapping two products, the system uses marked states to represent the desired final configuration (products at their swapped positions). For example, as shown in Listing 21 we specify that the new and final value of *storageHouse1A.holds* should be *Blue* and the value of *storageHouse2A.holds* should be *Red*. Rather than explicitly programming the sequence of operations needed to perform the swap, this approach allows the designer to simply specify what the end result should look like, and the synthesis algorithm automatically computes a supervisor that drives the system from the initial state to the marked goal state while respecting all safety constraints and maintaining system integrity. This marking-based approach offers greater flexibility and can handle more complex operational objectives without manually specifying execution sequences.

```
1     marked storageHouse1A.holds = Blue;
2     marked storageHouse3A.holds = Red;
```

Listing 21: Marked predicate approach.

8 Synthesized Controller

The core result of the SBE workflow presented in this thesis is the synthesized supervisory controller that governs the coordinated behavior of the demo system. This controller is automatically derived from the formal plant models and requirement specifications through supervisory control synthesis, which provides four fundamental guarantees: *(1) controllability* - the supervisor only disables controllable events, *(2) nonblocking* - the system can always reach a marked state, *(3) safety* - all forbidden behaviors specified in the requirements are prevented, and *(4) maximal permissiveness* - the supervisor allows all behaviors that satisfy the requirements. The resulting supervisor is correct-by-construction with respect to the specified requirements. The controller ensures that at any point during execution, only those controllable events that are consistent with the current scenario step, the homing and safety invariants, and the observed state of the plant are enabled. As a consequence, all valid executions produced by the controller satisfy the desired sequencing constraints, synchronization requirements, and safety properties without requiring additional runtime checks.

Although the synthesis algorithm used in the data-based synthesis tool that is employed by ESCET is fully automated, its effectiveness is strongly influenced by the ordering of variables in the underlying Binary Decision Diagrams (BDDs) representation. The data-based synthesis tool represents the system's state space symbolically using BDDs, encoding both discrete locations and data variables (such as integers and booleans) in a compact, symbolic form. This approach can handle large state spaces efficiently when the BDD representation remains compact, but performance is highly sensitive to variable ordering. This section aims to explain the performance optimization strategies applied to improve the efficiency of the synthesis process itself specifically, reducing the time and memory required to compute the supervisor. We focus primarily on optimizations for the data-based synthesis tool. In order to do this, first we need to give a brief explanation of BDDs.

8.1 Binary Decision Diagrams

Binary Decision Diagrams (BDDs) are data structures used to represent and manipulate boolean functions efficiently. In the context of supervisory control synthesis, BDDs enable the symbolic representation of large state spaces without explicitly enumerating every individual state. Instead of storing states as separate entities, BDDs encode sets of states as boolean functions over system variables.

A BDD is organized as a directed acyclic graph where each internal node represents a decision on a boolean variable, and the two outgoing edges represent the outcomes when that variable is true or false. The leaves of the graph represent the final boolean result (true or false).

The efficiency of BDD-based operations such as computing reachable states, checking transitions, or determining enabled events, depends critically on the variable ordering: the sequence in which boolean variables are tested along paths in the BDD. Variables that appear near the top of the ordering (tested first along every path) have a stronger influence on the BDD's structure than variables lower in the ordering. When variables that frequently interact or have strong dependencies are placed close together in the ordering, the BDD can exploit these relationships to create shared substructures, resulting in more compact representations. Conversely, poor variable orderings can cause BDDs

to grow exponentially, leading to excessive memory consumption and prohibitively long computation times during synthesis. This is particularly problematic for systems with many concurrent components, where a random ordering might interleave variables from unrelated subsystems, preventing the BDD from capturing their independent behaviors efficiently. Many heuristics for variable ordering exist, but this remains an area of active research [9]. Furthermore, some heuristics work better in certain cases while others perform better in different situations, making it difficult to know beforehand which heuristic yields the best performance for a given system.

8.2 Performance Improvements

In the context of this work, particular attention is paid to variables related to product ownership and spatial configuration, specifically variables with suffixes *.holds* and *.move_pos*, as we prioritize them by placing them at the top of the BDD variable ordering. These variables encode which component currently holds a product and the discrete position of movable elements, respectively. It is important to note that manual BDD variable ordering optimization represents advanced usage of ESCET and is not typically required for most applications. The default automatic variable ordering provided by ESCET is sufficient for many systems. However, for complex industrial systems with extensive state spaces and intricate component interactions such as the demo system presented in this thesis manual optimization can be critical to achieving tractable synthesis times. The work described here involved substantial experimentation and analysis to identify the optimal ordering strategy, requiring understanding of both the system's operational patterns and BDD performance characteristics. The rationale for prioritizing these variables stems from their central role in the production workflow. Product handovers between subsystems constitute the fundamental operations of the demo system. Each handover involves synchronized transitions that update multiple *.holds* variables simultaneously, with guards that depend on both the source and destination components being in specific positions (*.move_pos* values). These variables are frequently accessed and are also tightly coupled: a product can only be transferred when both the donating and receiving components are correctly positioned and in the appropriate state. For this reason, the BDD variable ordering is explicitly adjusted such that variables representing product possession (*.holds*) and positional state (*.move_pos*) are assigned higher priority in the ordering, meaning they are tested earlier in BDD decision paths. This reordering reduces unnecessary interleavings between unrelated variables and leads to more compact decision diagrams, as the BDD can exploit the repeated patterns of product handovers that dominate the system's behavior, as we will see later in Chapter 10.

9 Code generation

An essential objective of Synthesis-Based Engineering is to bridge the gap between formally synthesized controllers and their practical deployment on real automation systems. In this thesis, the supervisor synthesized from the *ScenarioTracker* specification is transformed into executable Java code and integrated with both the TNO-ESI demo system and the Digital Twin via an OPC UA based middleware layer as described in Section 4.2.1. This chapter describes how the synthesized controller is realized in software and explains the role of the principal Java classes that enable its execution.

Following synthesis, the ESCET toolchain produces a supervisor that encodes all admissible control decisions as derived from the plant models and formal requirements. Using CIF's code generation capabilities, the supervisor is translated into Java code that preserves the operational semantics of the CIF model, thereby ensuring that the generated controller enforces the specified behavior exactly as synthesized. The code generation process produces a Java class, named *SynthesizedController* that contains: state variables representing all discrete variables from the CIF model, logic for evaluating guards, updating state variables, and determining enabled events.

The generated controller does not directly interact with the physical system or digital twin. Instead, it operates as a decision-making component that determines which controllable actions may be executed at any point in time. To deploy this controller on the demo system, an additional integration layer is required. This layer is implemented in Java and consists of three primary classes: *InterfacedController*, *IHAL*, and *OpcUaService*.

The *InterfacedController* class extends the generated *SynthesizedController* class and serves as the main execution and coordination component. This class is responsible for running the synthesized supervisor continuously as shown in Listing 22.

Listing 22: Run method that executes the synthesized controller.

```
public void run() {
    System.out.println("Starting workflow execution...");
    double currentTime = 0.0;
    double timeStep = 0.1; // 100 ms per cycle

    // Run until workflow completes
    while (ScenarioTracker_currentStep_ <= FINAL_STEP) {
        execOnce(currentTime);
        currentTime += timeStep;
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt(); // Restore interrupt status
            throw new RuntimeException("Workflow execution interrupted", e);
        }
    }
    System.out.println("Workflow completed successfully!");
}
```

Furthermore this class also does the translation of CIF events into actual hardware

commands with proper parameters, and updating input variables based on command completion status, as shown in Listing 23.

Listing 23: Translation of CIF commands into hardware commands.

```
private void executeHardwareCommand(String eventName) {
    try {
        if (eventName.equals("homeWarehouse.c_start")) {
            opc.call(homeWarehouse, new Variant[0]);
            homeWarehouse_executed_ = true;
        }
        else if (eventName.equals("moveToStorage2A.c_start")) {
            Variant[] args = {
                new Variant(UShort.valueOf(STORAGE_2A.x)),
                new Variant(UShort.valueOf(STORAGE_2A.z))
            };
            opc.call(moveStacker, args);
            moveToStorage2A_executed_ = true;
        }
        else if (eventName.equals("stackerToUpPos.c_start")) {
            opc.call(stackerToUpPos, new Variant[0]);
            stackerToUpPos_executed_ = true;
        }
        else {
            throw new IllegalArgumentException("Unknown event name: " + eve
        }

    } catch (Exception e) {
        System.err.println("Error executing: " + eventName);
        e.printStackTrace();
        throw new RuntimeException("Failed to execute hardware command: " +
    }
}
```

Crucially, *InterfacedController* does not encode any task-specific behavior or sequencing logic. All decisions regarding the order and conditions under which commands are executed are determined exclusively by the synthesized supervisor inherited from *SynthesizedController*. This design preserves the declarative nature of the SBE approach, in which controller behavior is derived from formal models rather than implemented manually. The class does, however, provide the necessary infrastructure for coordinate mapping and parameter conversion. Coordinate mapping translates abstract CIF positions (such as *Storage2A*) into actual hardware coordinates (such as $x = 116$, $z = 68$), as can be seen in Listing 24.

Listing 24: Mapping of the discrete positions to coordinates in *x, z* plane.

```
// Storage position coordinates for stacker commands
private static class StoragePosition {
    final int x, z;
    StoragePosition(int x, int z) {
        this.x = x;
        this.z = z;
    }
}

private static final StoragePosition STORAGE_1A =
    new StoragePosition(116, 16);
private static final StoragePosition STORAGE_2A =
    new StoragePosition(116, 68);
private static final StoragePosition BELT =
    new StoragePosition(3, 115);
```

Interaction with the physical system is abstracted through the IHAL, which is represented in Java by the 'IHAL' class. This class provides a uniform interface for invoking system commands and receiving execution results, independent of of whether the target is physical hardware or the digital twin simulation, as shown in Listing 25.

Listing 25: IHAL Class.

```
public final class IHAL {
    public static class Method {
        public final NodeId callNode;
        public final NodeId completedNode;
        public final String serverKey;

        public Method(String serverKey, NodeId callNode, NodeId completedNo
            this.serverKey = serverKey;
            this.callNode = callNode;
            this.completedNode = completedNode;
        }
    }
}
```

The IHAL abstraction provides two critical benefits. First, it ensures hardware independence, meaning that the same controller code works with both the physical demo system and the digital twin without modification. Second, it maintains consistent semantics by ensuring that the command execution model (call, wait, completion) matches the CIF specification exactly.

The actual communication interface with both the physical demonstration system and its digital twin counterpart is implemented through the *OpcUaService* class. This class serves as an abstraction layer that encapsulates all OPC UA related functionality, thus providing a clear separation between the application logic and the industrial

communication protocol. This class handles the initialization of the connection with the OPC UA server endpoint through *@PostConstruct* annotations. The shutdown procedure is handled through *@PreDestroy* annotations (shown in Listing 26) to ensure reliable communication throughout the application lifecycle.

Listing 26: Connection establishment and termination.

```
private OpcUaClient client;

@PostConstruct
public void connect() throws Exception {
    client = OpcUaClient.create("opc.tcp://localhost:4841");
    client.connect().get();

    System.out.println("Connected to OPC UA server");

    subscription = client.getSubscriptionManager()
        .createSubscription()
        .get();
}

@PreDestroy
public void disconnect() throws Exception {
    if (client != null) {
        client.disconnect().get();
        client = null;
    }
    System.out.println("Disconnected from OPC UA server");
}
```

Furthermore, the service implements subscription management and event monitoring. Subscription callbacks are registered for system nodes, particularly completion indicators denoted by **Completed/ActiveState/Id* nodes, corresponding to operational tasks such as robot movements, conveyor actions, oven processes, and stacking operations. By subscribing to these node, the service receives notifications whenever an operation becomes active or has been completed.

The synthesized controller, the execution logic in *ManualSynthesis*, and the communication services operate together. The synthesized controller (implemented in *ControllerSynthesis*) evaluates the current state and determines which controllable events are enabled based on the formal requirements. Then the *ManualSynthesis* class intercepts enabled events via the *infoEvent* callback and executes the corresponding hardware command. When the controller decides to trigger a controllable event such as *moveToStorage2A.c_start*, the *ManualSynthesis* class translates this abstract event into a concrete hardware command with the appropriate parameters. The command is transmitted through *OpcUaService* to the physical system or digital twin via OPC UA. The service invokes the appropriate OPC UA method on the target system and begins monitoring for completion. Completion status is detected via OPC UA subscriptions that monitor specific completion nodes on the OPC UA server. When a command completes

successfully, the *OpcUaService* notifies *ManualSynthesis*, which updates the controller's input variables by setting the corresponding *executed* flags to true, which is picked up by the generated code and used to update the model state and determine next events that can be taken.

10 Demonstration and Results

This section presents the practical demonstration of the synthesized supervisory controller operating on the TNO-ESI demo system. Multiple demonstration scenarios are conducted to illustrate the capabilities, scalability, and flexibility of the Synthesis-Based Engineering approach in managing automated production processes.

The first scenario represents a complete production workflow, as previously described in Section 7.2. In this demonstration, a single product is transferred through every stage of the factory from its initial position in the warehouse, through the processing stations, to the sorting area, and back to the warehouse again. This scenario validates that the synthesized controller can correctly coordinate the sequence of operations required for end-to-end product handling in a standardized manufacturing process.

The second scenario extends the concept of concurrent product handling by moving two products through the complete production workflow simultaneously. Similar to the first scenario, both products follow the entire manufacturing process from warehouse retrieval, through processing stations (baking and polishing), to final sorting and then back to their initial places.

The third scenario highlights the system's ability to orchestrate more complex and dynamic operations by simultaneously managing the movements of two products along different production paths. This "swap" demonstration begins with two items located in the warehouse: a red product at storage position 1A and a blue product at position 3A. The objective is to exchange their locations by moving the red product from 1A to 3A and the blue product from 3A to 1A in parallel. Unlike the first scenario, which follows a single linear workflow, this scenario requires the supervisory controller to coordinate two concurrent sequences of actions, ensuring that shared resources are accessed safely and that no conflicts occur during simultaneous operations. This demonstration confirms that the SBE methodology supports not only predefined and sequential workflows but also dynamic, multi-product scenarios. The implementation of this scenario is achieved using a dedicated scenario automaton, structurally similar to the one employed for the single-product case but extended to model two concurrent process flows. As a result, the automaton contains a single location with guards on its edges that control the execution sequence. At runtime, this results in approximately twice the number of states and transitions in the controlled system's state space compared to the individual scenario, reflecting the increased complexity of coordinating two interleaved task sequences. However, this estimate assumes a fixed, manually defined interleaving of the two scenarios. If synthesis is allowed to explore additional valid interleavings such as different orderings of independent operations between the two scenarios the actual state space could grow significantly larger, as the supervisor considers all safe interleavings of concurrent actions. Despite this increased complexity, the formal guarantees of safety and correctness provided by the synthesis process are preserved.

The fourth demonstration showcases the system's ability to swap two products using the marked predicate approach. In this scenario, the goal is to "swap" the products by explicitly defining in which positions certain colored product should be at the end of the production workflow. As mentioned in Section 7.3, the end goal is to switch the products from *storageHouse1A* and *storageHouse3A*, and then let the synthesis do the rest.

In order to evaluate the impact of variable ordering and requirement specification approaches on synthesis performance, several experiments are conducted with different system configurations. First we run the synthesis algorithm with default settings, in

particular the heuristic variable ordering of the CIF synthesis tool. Then we apply our custom variable ordering, as described in Section 8.2. Table 1 presents the synthesis times for different combinations of system models and requirement specifications. In this table the "TIMEOUT" key word specifies that a controller could not be achieved using the synthesis process after 60 minutes.

Requirement Type	**Var. order**	**Synthesis Time**
One product loop (scenario)	Heuristics	50 seconds
Two product loop (scenario)	Heuristics	200 seconds
Two product swap (scenario)	Heuristics	TIMEOUT
Two product swap (marked predicate)	Heuristics	TIMEOUT
One product loop (scenario)	Custom	1 second
Two product loop (scenario)	Custom	3 seconds
Two product swap (scenario)	Custom	11 seconds
Two product swap (marked predicate)	Custom	TIMEOUT

Table 1: Synthesis times for different system configurations and requirement specifications.

The results demonstrate that variable ordering significantly impacts synthesis performance in larger systems, while the choice between scenario-based requirements and marked predicates can dramatically affect computational feasibility. The timeout observed for the marked predicate approach in the reduced system indicates that certain requirement formulations can make synthesis hard to deal with, even for models like the discussed demo system. Further investigation is needed to determine why exactly the marked predicate approach doesn't scale and whether and how this can be resolved.

11 Conclusions and Future work

This section explicitly addresses each sub-research question posed at the beginning of this thesis and reflects on the lessons learned throughout the project.

SRQ1: How can we best model the plants and requirements of the TNO-ESI demo system? The TNO-ESI demo system is best modeled using a hierarchical approach that separates concerns between commands, physical components, and coordination logic. We developed a generic *Command* automaton definition that captures the execution semantics of the IHAL interface, which is then instantiated for each specific command. Component-specific plant automata are created to model the operational behavior and state of each physical element (warehouse stacker, gripper, processing stations, sorting line). Requirements are partitioned into safety requirements, which define forbidden behavior, and scenario requirements, which specify desired production workflows. This part of the project is very enjoyable, but at the same time challenging as a lot of attention needs to be payed on how the physical behavior is translated into discrete states and events.

SRQ2: How can we best develop a simulation model for the TNO-ESI demo system to validate the plants and requirements? The simulation model is developed by extending the CIF plant models with visualization capabilities and discrete-event simulation logic. We implement a graphical simulation model that provides real-time visual feedback of component states, product locations, and command execution. The simulation proves invaluable for validating both individual component behaviors and integrated system operations before attempting synthesis or hardware deployment, ensuring consistency between simulated and physical system behavior. Developing the simulation model is one of the most enjoyable and rewarding aspects of the project. The immediate visual feedback makes it easy to identify modeling errors and validate component interactions. The simulation environment serves as an essential debugging tool throughout the development process, allowing rapid iteration without requiring access to the physical hardware.

SRQ3: How can we efficiently synthesize a supervisor for the TNO-ESI demo system? Efficient synthesis for the demo system requires careful attention to state-space management and BDD variable ordering optimization. Through systematic experimentation, we identify that prioritizing variables related to product ownership *(.holds)* and spatial configuration (*.move_pos*) at the top of the BDD variable ordering dramatically improves synthesis performance. Additionally, partitioning the system into subsystems and carefully structuring requirement automata to avoid unnecessary state-space expansion proves critical. For complex scenarios, we find that explicitly constraining the command sequences through scenario automata reduces the synthesis complexity compared to purely marking-based approaches. Controller synthesis is the most technically challenging aspect of the project. Understanding how modeling choices impact synthesis performance requires significant trial and error, especially if one is not experienced in this aspect. The relationship between model structure, variable ordering, and BDD size is non-intuitive, and optimizing synthesis often feels like solving a puzzle with many pieces.

SRQ4: How can we best connect the synthesis result for the TNO-ESI demo system to the digital twin and the physical system? Connection to both the digital twin and physical system is achieved through code generation targeting the IHAL interface abstraction layer. We develop a Java-based runtime architecture that

integrates the synthesized supervisor with OPC UA communication for hardware control. The key insight is to generate code that operates at the level of IHAL commands rather than directly manipulating hardware, ensuring portability between simulation and physical deployment. The generated supervisor code includes all hardware-specific details encapsulated within the IHAL implementation. Code generation and integration with IHAL is surprisingly straightforward and enjoyable. The clean separation between supervisor logic and hardware interface means that once the code generation framework is established, deploying it to the Digital Twin or the physical Demo System does not require any major changes. All we need to do is to create "glue code" that makes the generated code to work with the IHAL interaface.

SRQ5: What demonstrations can we create to best showcase the value of the SBE methodology using the TNO-ESI demo system? We create multiple demonstrations that showcase different aspects of the SBE methodology. The primary demonstration involves a complete production workflow that retrieves products from storage, processes them through multiple stations, and sorts them by color, all controlled by a synthesized supervisor that guarantees safety and nonblocking behavior. Additional demonstrations include scenario variations (single vs. multiple products).

While the current work successfully demonstrates the application of Synthesis-Based Engineering to the TNO-ESI demo system, several avenues for future research and development have been identified. The two-product swap scenario presented in the previous chapter explicitly specifies the sequence of steps needed to achieve the swap. This approach, while functional, has significant limitations. The scenario automaton must be manually designed for each specific swap operation, and extending it to handle swaps involving more than two products becomes increasingly complex. An alternative approach that is explored during this research involves the use of marked states and reachability-based synthesis. This declarative approach showcases synthesis at its best: the designer states the objective, and the algorithm derives the control logic, fundamentally shifting the paradigm from procedural "how-to" specifications to declarative "what-to-achieve" requirements.

However, preliminary investigations revealed significant scalability challenges with this method. When attempting to synthesize supervisors for swap scenarios involving marked states, the synthesis process encountered computational difficulties related to the size of the state space that must be explored. The freedom that makes this approach elegant allowing synthesis to consider all possible interleavings and execution path also dramatically expands the search space, leading to state explosion issues that exceeds available computational resources for the demo system's complexity.

12 References

References

[1] *Automata*. https://cs.stanford.edu/people/eroberts/courses/soco/projects/2004-05/automata-theory/basics.html.

[2] D. A. van Beek et al. "CIF 3: Model-Based Engineering of Supervisory Controllers". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014)*. Ed. by E. Ábrahám and K. Havelund. Vol. 8413. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2014, pp. 685–700. DOI: 10.1007/978-3-642-54862-8_48.

[3] *Eclipse ESCET – CIF Language*. https://eclipse.dev/escet/cif/. Eclipse Foundation.

[4] W. Fokkink and M. Goorden. "Offline Supervisory Control Synthesis: Taxonomy and Recent Developments". In: *Discrete Event Dynamic Systems* 34 (2024), pp. 605–657.

[5] W. J. Fokkink et al. "Eclipse ESCET™: The Eclipse Supervisory Control Engineering Toolkit". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2023)*. Ed. by S. Sankaranarayanan and N. Sharygina. Vol. 13994. Lecture Notes in Computer Science. Cham: Springer, 2023. DOI: 10.1007/978-3-031-30820-8_6.

[6] W. J. Fokkink et al. "Supervisor Synthesis: Bridging Theory and Practice". In: *Computer* 55.10 (2022). Article 9903886. DOI: 10.1109/MC.2021.313493.

[7] Wan Fokkink et al. "Synthesis-based engineering of supervisory controllers". In: *Mikroniek* 2023 (2023).

[8] A. G. C. Gonzalez et al. "Supervisory Control-Based Navigation Architecture: A New Framework for Autonomous Robots in Industry 4.0 Environments". In: *IEEE Transactions on Industrial Informatics* 14.4 (2018), pp. 1732–1743. DOI: 10.1109/TII.2017.2788079.

[9] Dennis Hendriks et al. *Overview and Performance Evaluation of Supervisory Controller Synthesis with Eclipse ESCET v4.0*. Tech. rep. SSRN, 2024. DOI: 10.2139/ssrn.4947024.

[10] Changmin Hong and Tae-Eog Lee. "Modeling, Simulation and Supervisory Control of Semiconductor Manufacturing Cluster Tools with an Equipment Front-End Module". In: *Proceedings of the 16th IEEE Conference on Automation Science and Engineering (CASE)*. Piscataway, NJ: IEEE, 2020, pp. 703–709. DOI: 10.1109/CASE48305.2020.9216790.

[11] *Inkscape Learning Resources*. https://inkscape.org/learn/. Accessed: 2025-03-08.

[12] Tom Korssen et al. "Systematic Model-Based Design and Implementation of Supervisors for Advanced Driver Assistance Systems". In: *IEEE Transactions on Intelligent Transportation Systems* 19.2 (2018), pp. 533–544. DOI: 10.1109/TITS.2017.2776354.

[13] Jonas Krook et al. “Modeling and Synthesis of the Lane Change Function of an Autonomous Vehicle”. In: *Proceedings of the 14th Workshop on Discrete Event Systems (WODES)*. Vol. 51. IFAC-PapersOnLine. Amsterdam: Elsevier, 2018, pp. 133–138. DOI: 10.1016/j.ifacol.2018.06.291.

[14] Genki Miyauchi, Yara Kaszubowski Lopes, and Roderich Groß. “Multi-Operator Control of Connectivity-Preserving Robot Swarms Using Supervisory Control Theory”. In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. Piscataway, NJ: IEEE, 2022, pp. 6889–6895. DOI: 10.1109/ICRA46639.2022.9812242.

[15] F. F. H. Reijnen et al. “Modeling for Supervisor Synthesis – A Lock-Bridge Combination Case Study”. In: *Discrete Event Dynamic Systems* 30 (2020), pp. 499–532. DOI: 10.1007/s10626-020-00314-0.

[16] Michel A. Reniers and Jolanda M. van de Mortel-Fronczak. “An Engineering Perspective on Model-Based Design of Supervisors”. In: *Proceedings of the 14th Workshop on Discrete Event Systems (WODES)*. Vol. 51. IFAC-PapersOnLine. Amsterdam: Elsevier, 2018, pp. 257–264. DOI: 10.1016/j.ifacol.2018.06.310.

[17] *Scalable Vector Graphics (SVG) 1.1 Specification*. https://www.w3.org/TR/SVG11/. W3C Recommendation.

[18] *Training Factory Industry 4.0 24V*. https://www.fischertechnik.de/en/products/industry-and-universities/training-models/554868-training-factory-industry-4-0-24v. Accessed: 2026-01-10. fischertechnik GmbH, n.d.

[19] *Unity*. https://unity.com/. Accessed: 2025-12-29. Unity Technologies, 2025.